《C++ Templates》读书笔记

整理日期：20120610

主页：http://chuanqi.name

E-mail：chuanqi.tan@gmail.com

这是一本精通C++模板技术的必读书，详细透彻的讲解和仔细的笔记

欢迎交流、指导；本文采用"CC BY 2.5"许可协议

By 谭川奇

前言

这本书并不是一本容易学习的书，仔细的看懂它、理解它会让一般的新手汗流浃背，比如我。但是，它值得你去这么做！作为 C++模板技术的两大经典之一，它提供了你精通 C++模板技术所必需的基本知识和思维方式。两位作者分别从模板的编译器实现角度和模板的实用技术角度给出了优美的讲解，这样对于我们从多种角度、深入的理解模板技术是大有裨益的。

但愿本笔记能为你的学习提供一些帮助！

谭川奇 2012.06.10 于北京
chuanqi.tan@gmail.com

本文采用的约定

蓝色：概念、讨论主题

下划线：值得注意的内容

紫红色：比较重要的地方

红色：必须理解的地方

绿色：一些评论、注解

斜体：一些总结性的话

粗体：对上述所有标记的加强、警示作用

目录

第一章：关于本书......4
第一部分：基础......5
第二章：函数模板......5
第三章：类模板......7
第四章：非类型模板参数......8
第五章：技巧性基础知识......8
第六章：模板实战......11
第七章：模板术语......14
第二部分：深入模板......16
第八章：深入模板基础......16
第九章：模板中的名称......21
第十章：实例化......25
第十一章：模板实参演绎......29
第十二章：特化与重载......31
第十三章：未来的方向......35
第三部分：模板与设计......40
第十四章：模板的多态威力......40
第十五章：trait 与 policy 类......41
第十六章：模板与继承......45
第十七章：metaprogram......49
第十八章：表达式模板......52
第四部分：高级应用程序......54
第十九章：类型区分......54
第二十章：智能指针......61
第二十一章：tuple......65
第二十二章：函数对象和回调......67
附录 A：一处定义原则......71
附录 B：重载解析......73

第一章：关于本书

1. 本书的组织结构：
    1. 第一部分：模板的基础知识；
    2. 第二部分：深入理解模板技术的许多细节；
    3. 第三部分：介绍实用的编码技术；
    4. 第四部分：注重于如何把模板技术应用到具体的应用程序中。
    5. 总体感觉上，前 2 部分是从编译器实现的角度去讲解模板；而后 2 部分则是从模板的实用技术上去讲解模板。
2. const 位置的选择，倾向于使用 int const 而不是 const int，主要有两个原因：
    - 首先，针对问题什么是不变的有着更好的一致性，int const 提更好的回答恒定不的部分始终指的是 const 限定符前面的内容。
    - 其次：在使用一种常用的 typedef 替换之后，const int 的含义有时候会发生变化，而 int const 则不会，下面是一个二义性的例子。

      首先进行了这个定义：typedef char* CHARS; //CHARS 是一个**指针类型**

      typedef const CHARS CPTR; //希望表达：指向 char 类型的常量指针

      展开后 => typedef const char* CPTR; //实际意思：指向常量 char 类型的指针

      如果使用 const 在后的格式

      typedef CHARS const CPTR; //希望表达：指向 char 类型的常量指针

      展开后 => typedef char* const CPTR; //实际意思：指向 char 类型的常量指针
    - 但是，现在看到的几乎所有的代码都是用的 const int 格式，如果我写成 int const 会不会造成他人难以理解？

第一部分：基础

为什么要使用模板：使用模板能激励你写更复杂更好的算法，因为它可以在以后被重用。使用模板的重用方式保留了类型检查这个优点，并且避免了使用愚蠢的"预处理"机制（可以这么理解：模板是比宏更好的预处理。但模板并不是一种真正的预处理，而是一种介于代码和预处理之间的机制）。

第二章：函数模板

- 函数模板提供了一个函数家族供使用，示例如下：

```
template<typename T>
inline T const& max(T const &a, T const &b){
  return a < b ? b : a;
}
```

- class 与 typename 的区别：

  从语义上讲，这里的 class 和 typename 是等价的。然而，class 这种用法往往会给人误导（这里的 class 并不意味着只有类才能被用来替代 T，事实上基本类型也可以）；因此对于引入类型参数的这种用法，应该尽量使用 typename。

  同时要注意，在这里 struct 不能替代 class 的。
- 模板的工作原理，并不是把模板编译成一个可以处理任何类型的单一实体；而是对于实例化模板参数的每种类型，都从模板产生出一个不同的实体。
- 模板实例化过程：在实例化时，模板被编译了两次，分别发生在
  - 实例化前，先检查模板代码本身，查看语法是否正确；在这里会发现错误语法，如遗漏分号等。
  - 在实例化期间，检查模板代码，查看是否所有的调用都有效。在这里会发现无效的调用，如该实例化类型不支持某些函数调用等。
- 由于 C++中使用的是静态模板的机制，所以当使用函数模板，并且引发模板实例化的时候，编译器（在某时刻）需要查看模板的定义。这就不同于普通函数中编译和之间的区别，因为对于普通函数而，只要有该函数的声明（不需要定义），就可以顺利通过编译。因为在这里需要的是定义，所以可以考虑在头文件内部实现每个模板以使用编译器能够顺利的找到模板的定义。
- 函数模板的实参演绎：当我们为某些实参调用一个诸如 max()的模板时，模板参数可以由我们所传递的实参来决定，这称为实参演绎，它使用可以像调用普通函数那样调用函数模板。

  绝不允许自动类型转换：在实参演绎时，不允许任何的自动类型转换，每个 T 都必须正确匹配。

  通常而言，你必须指定"最后一个不能被隐式演绎的模板实参之前的"所有实参类型。

  然而，模板的实参演绎并不适合返回类型（可以把演绎看成是重载解析的一部分--重载解析是一个不依赖于返回类型选择的过程，唯一的例外就是转型操作符成员的返回类型）。
- 在函数模板内部，不能指定缺省的模板实参。这一点和类模板是不同的，类模板是可以指定缺省实

参的。

C++11 里已经可以正确的指定函数模板的缺省参数了。

- C++里并没有提供一种指定并且选择一个"最强大类型"的途径（然而，可以通过使用一些 tricky 模板编程来提供这个特性，书后面有介绍了）。
- 和普通函数一样，函数模板可以被重载。不止可以被重载，还可以进行特化。

```
template<typename T1>
int Do(T1 t){
  return 0;
}
template<>
int Do<double>(double t){
  return -1;
}
int Do(string const &str){
  return 999;
}
int main() {
  cout << Do(3) << endl;
  cout << Do(10.0) << endl;
  cout << Do(std::string("abc")) << endl;
  cout << Do<>(std::string("abc")) << endl;
  return 0;
}
```

输出为：0，-1，999，0

重载规则优先调用已经存在的非模板函数，而不会从模板产生出一个实例，所以 `Do(std::string("abc"))`输出 999；

可以显式地指定一个空的模板实参列表，这个语法好像告诉编译器：只有模板才能来匹配这个调用，而且所有的模板参数都应该根据调用实参演绎出来，所以 `Do<>(std::string("abc"))`输出 0；

- 模板是不允许自动类型转换的（必须牢记于心），但是普通函数可以进行自动类型转换。

  所以像 max('a', 42.7)这样的调用，就会直接使用普通函数的版本 max(int, int)，并通过自动类型转换实现调用。
- 一般而言，当重载函数模板时，最好只改变那些需要改变的内容，也就是说应该把改变限制在：显式地指定模板参数+改变参数数目。否则，会出现诡异的非预期的结果（如从传引用改变为传值）。
- **牢记一条首要规则**：一定要让函数模板的所有重载版本的声明都位于它们被调用的位置之前（一般都放在同一个头文件中）。

第三章：类模板

1. 类模板的声明：

```
template<typename T>
class Stack{
 Stack(Stack<T> const &);
 ~Stack();
 Stack<T> operator=(Stack<T> const &);
};
```

   区分两种写法：类的**类名** Stack、类的**类型** Stack<T>

   当在声明中需要使用类的类型时，你必须使用 Stack<T>（大部分情况）；然而当使用类名而不是类的类型时，就应该只用 Stack（比如指定类的名称、构造和析构函数名）。

2. vector 的 pop_back()方法只是删除末尾的元素，并没有返回该元素；之所以如此是充分考虑了异常安全性，因为要实现"一个绝对异常安全并且返回被删除元素的 pop()"是不可能的。

3. 注意，只有那些被调用的成员函数，才会产生这些函数的实例化代码。对于类模板，成员函数只有在被使用的时候才会实例化。

   显然，这样可以节省空间和时间；另一个好处是对于那些"未能提供所有成员函数中所有操作的"类型，你也可以使用该类型来实例化类模板，只要对那些"未能提供某些操作的"成员函数，模板内部不使用就可以。

   而且现在的 C++标准要求编译器要尽可能的延迟实例化的时机。

4. 静态成员的无条件实例化：如果类模板中含有静态成员，那么用来实例化的每种类型，都会实例化这些静态成员。

5. C++中的类型定义（typedef）只是定义了一个"类型别名 alias"，而不是一种新的类型，所以对于 typedef Stack<int> IntStack; 它们实现上是完全等价的，并且可以相互赋值。

6. 和函数模板的重载类似，通过特化类模板，你可以优化基于某种特定类型的实现，或者克服某种特定类型在实例化模板时所出现的不足（例如该类型没有提供某种操作，要以其它的方法来实现）。

   另外，如果要特化一个类模板，你还要特化该类模板的所有成员函数。虽然也可以只特化某个成员函数，但这个做法并没有特化整个类，也就没有特化整个类模板。

   特化一个类模板：

```
template<>
class Stack<std::string>{
    ......
}
```

   注意：*特化的实现可以和基本类模板的实现完全不同，连接口都可以完全不同(没有任何限制）。*

7. 也可以局部特化类模板，但使用含有局部特化的类模板时，会**自动选取最匹配的那个模板**使用。

8. 对于类模板，可以为模板参数定义缺省值；这些值就被称为缺省模板实参；而且，它们还可以引用之前的模板参数。

   在 C++11 之前，只有类模板才能有缺省的模板实参，但是 C++11 之后，函数模板也能有缺省的实参了。

第四章：非类型模板参数

1. 模板参数并不局限于类型，普通值也可以作为模板的参数，同样值类型也可以指定缺省值。

   非类型的类模板参数：

   ```
   template <typename T, int MAXSIZE=66>
   class Stack{
      T elems[MAXSIZE];
   }
   ```

   非类型的函数模板参数：

   ```
   template<typename T, int VAL=6>
   T addValue(T const &x){
     return x + VAL;
   }
   ```

2. **非类型模板参数是有限制的：**通常而言，它们可以是常整数（包括枚举值）或者指向外部链接对象的指针。（是否因为它们的大小是固定的）

   浮点数、类对象和内部链接对象是不允许作为非类型模板参数的。

3. 书上说浮点数作为模板参数的实现是没有技术上的障碍的，但是 C++11 依然不支持使用浮点数作为模板参数（gcc4.6.3 提示错误：double 不是一个有效的模板常量参数类型）。

   我想：使用非类型作为模板参数是没有太大的实际意思的，这种变化完全可以使用构造函数的参数来进行模拟；之所以支持整数类型的模板参数，可能是由于以前数组的个数必须要求为编译常量所引起的吧。

   非类型的模板参数有什么我不知道的用途呢？

第五章：技巧性基础知识

1. 在 C++标准化过程中，引入关键字 typename 是为了说明：模板内部的标识符可以是一个类型（而不只是类 class）。譬如下面的例子：

   ```
   template <typename T>
   class MyClass{
   ```

typename T::SubType *ptr;

}

在这个程序中，第 2 个 typename 被用来说明：SubType 是定义于类 T 内部的一种类型。因此，ptr 是一个指向 T::SubType 类型的指针。

如果不使用 typename，SubType 就会被认为是一个静态成员，那么它应该是一个具体变量或对象，于是表达式：T::Subtype * ptr 会被看作是类 T 的静态成员 SubType 和 ptr 的乘积。

通常：当某个依赖于模板参数的名称是一个类型时，就应该使用 typename 明确的标识！

2. **奇怪的.template 构造**：

```
template<int N>
void printBitset(bitset<N> const &bs){
    string str = bs.template to_string<char, char_traits<char>,
allocator<char> >();
    cout << str;
}
```

这种语法看似很奇怪，其实也是出现过的，在显示进行操作符时也用过类似的语法：

B b = a.operator B();

显式的告诉编译器进行一个从 A 到 B 的类型转换。

同样，这里的意思就是显式地告诉编译器后面将出现一个成员模板函数 to_string，否则编译器会无法识别 to_string 之后的"<"是模板实参的开始符号而非小于号。在编译器无法自己判断时使用.template 来辅助编译器进行语法分析。

总之：只有当该前面存在依赖于模板参数的对象(bs)时，并且后面紧跟着的也是模板调用(.to_string())时，编译器将无法自动知道后面的调用是一个模板调用，所以我们需要使用.template 辅助编译器进行语法分析（*参见第 9 章*）。

3. **具有依赖型基类的类模板符号解析时的怪异点**：对于那些在基类中声明，并且依赖于模板参数的符号（函数或者变量等），你应该在它们前面使用 this->或者 Base<T>::。

template <typename T>

class Derived : public Base<T> {…}

如果希望完全避免不确定性，你可以（使用诸如 this->和 Base<T>::等）限定（模板中）所有成员的访问（*参见第 9 章*）。

因为 C++标准规定非依赖型名称不会在依赖型基类中进行查找。（原因：*非依赖名称的查找是在模板的定义时而非实例化时，这个时候依赖基类的定义也是不完全的。加上 this->时就把名称变成了依赖的，于是名称查找就推迟到了实例化时，这时基类已完全，所以可以去基类查找*）

4. 成员模板：类成员也可以是模板。嵌套类和成员函数都可以作为模板。

template <typename T>

```
class Stack{
  template<typename T2>
  Stack<T>& operator=(Stack<T2> const &);
};
```

定义实现如下：（定义内部模板的语法：两层 template<...>）

```
template <typename T>
template <typename T2>
Stack<T>& Stack<T>::operator=(Stack<T2> const &op2){
   //......
}
```

需要注意的一点：模板赋值运算符并没有取代缺省赋值运算符。对于相同类型 Stack 之间的赋值，仍然会调用缺省赋值运算符，所以需要两个版本的 operator=。

5. 模板的模板参数：把类模板作为模板参数是很有用的，我们称之为"模板的模板参数"。借助"模板的模板参数"，可以实现只指定容器的类型而不需要指定所含元素的类型。

```
template<typename T,
  template<typename ELEM,
  typename ALLOC = std::allocator<ELEM>> //模板的模板实参必须精确匹配，匹配时并不会考虑"模板的模板实参"的缺省模板实参，必须显式指出这个 ALLOC！
  class CONT>      //模板的模板参数只能使用关键字 class
class Stack{
public:
  Stack(){}      //必须定义，因为声明变量 s 时构造函数必须被调用
  Stack(Stack<T, CONT> const &);//不必定义，"只有哪些被调用的类成员会会被实例化"
  ~Stack(){}     //必须定义，因为声明变量 s 时析构函数必须被调用
  Stack<T, CONT> operator=(Stack<T, CONT> const &);//不必定义，同上
private:
  CONT<T> elems;
};
```

这样，只需要这样使用 `Stack<int, vector> stack;`不必指出 vector 的实参为 int，模板会自动扩展。

支持模板的模板参数是 C++98 标准中的要求，但是时至今日，虽然 gcc4.6.1 可以正确的编译，但是 eclipse cdt 是居然还智能提示语法错误（代码中红色带下划线部分）。

6. 零初始化：对于模板中的变量，应该对它们进行显式的初始化。

```
T x = T();
```

或者在缺省的构造函数中：

MyClass() : x() {...}

7. 数组到指针的退化：实参演绎过程中，当且仅当参数不是引用时，会出现数组到指针的类型转换（称之为 decay，普通的函数调用也经常遇见数组退化为指针的情况，属于 C 语言的内容）。
C++对象模板里有深入的解释，数组的大小编译器会偷偷的放在某个地方，但是当作为参数传递时不会把这个数组的大小也一起传过去，于是就退化为了指针传递过去。

```
template <typename T>
inline T const & max(T const &a, T const &b){ //这里是引用，不会 decay
 return a < b ? b : a;
}
std::string s;
::max("apple", "peach");     //OK：相同类型的实参 char[5]
::max("apple", "tomato");   //ERROR：不同类型的实参 char[5], char[6]
::max("apple", s);            //ERROR：不同类型的实参 char[5], std::string
```

一个试探的解决方案是将 max 变为：

```
template <typename T> inline T max(T a, T b);
```

通过在非引用实参时的 decay 将不同长度的字符数组都 decay 成了字符指针，这样第 1 个 ERROR 就能通过编译，但实际上 max 里仅仅比较指针地址（语法正确、语义错误）。
真正的解决方案是把 char[n]全部转换成 std::string 类型，这也体现标准库 string 的优点：::max(std::string(“apple”), std::string(“peach”));
标准库中的很多函数都是最普通的引用传递（如 std::max 等），需知道它们的局限性。

第六章：模板实战

- *从某种意义上讲，模板是位于宏和普通（非模板）声明之间的一种构造。*
- 由于 C++采用静态模板的原理，使用编译器在编译时必须知道模板的完整定义，决不能推迟到链接时刻。所以使得原始的"编译器+链接器"模型带来了挑战，因此需要使用其它的方法来组织模板代码，这些方法是包含模型、显式实例化和分离模型。
  C++11 标准已经否决了分离模型，所以以后的模板就只用包含模型！
- 在大多数情况下，都应该使用包含模型（就是说把所有的模板代码都放在头文件中）。~~但是分离模型是 C++专门为解决这种情况而设计的，所以在编译器普遍支持之后应该考虑使用这种模型。~~
- 包含模板大大的增加了编译复杂程序所耗费的时间。
- 通过把模板声明代码和模板定义代码放在不同的头文件中，你可以很容易地在包含模型和显式实例化之间做出选择。

- ~~C++标准为模板定义了一个分离的编译模型（使用关键字 export）。然而，该关键字的使用还没有普及，很多编译器不支持(C++11 中被抛弃)。~~
- **显式实例化**：实例化类模板时会同时实例化它的所有类成员，而且对于每个不同的实体，在一个程序中最多只能有一个实例化体，并且也就不能有显式特化了。

```
template class Stack<int>;          //显式实例化类模板
template Stack<int>::Stack();       //显式实例化类模板的成员函数
template void PrintType<double>(double const &);     //显式实例化函数模板
```

同样，也**不推荐使用显式实例化**，因为它有的一个显著缺点：必须手动跟踪每一个需要实例化的实体，因为一个程序中不允许有多份的实例化实体。

它的优点就是避免了包含庞大的头文件的开销，更可以把模板定义的源文件封装起来（但是模板的使用者就不能够使用其它类型进行额外的实例化了，是的！）。

显式实例化正确的使用时机：*而且这种封装模板的源文件，只提供.h 文件，是显式实例化机制的唯一正确的、可移植的使用时机！*

- **为什么显式实例化可以把模板的定义源文件封装起来呢？**

因为编译器编译模板的原理是这样的：与普通的非模板类一样，只需要有类的接口（template.h），就可以编译代码了，但是在链接阶段，链接器就必须找到所有使用的成员函数的确切定义（template.cpp）。对于普通类定义这没问题，但是对于模板类这些成员函数并没有定义（只存在一个用来生成定义的模板），需要编译器根据需求去实例化定义。如果把模板的定义源文件封装起来了，那么编译器连模板都看不到，当然就没办法实例化相应的定义了。不过如果在 template.cpp 中显式实例化了部分类型的成员函数，那么就不需要编译器去实例化了，仅仅告诉链接器去链接相应的符号就可以了。

下面写一个例子一下就明白了：

>>> stack.h

```
template<typename T>
class Stack {
 public:
        Stack();
};
```

>>> stack.cpp

```
template Stack<int>::Stack();   //显式实例化了针对 int 类型的构造函数

template<typename T>
Stack<T>::Stack() {}
```

>>> main.cpp

```
#include "stack.h"
Stack<int> s1;       //OK
Stack<double> s2;    //ERROR
```

编译器看到了 Stack<int> s1;就需要去找 Stack<int>::Stack()来构造一个对象，但是由于编译器在本编译单元 man.cpp 中看不到构造函数的定义，也就无法自动去实例化，只能做个标记这个构造函数需要链接器去链接符号，但是由于在另外的编译单元 stack.cpp 中显式实例化了 template Stack<int>::Stack()，于是链接器顺利链接成功。

同样的道理，对于 Stack<double> s2，编译器同样也只能做标记，但是这次链接器无法在其它的编译单元发现相应的符号，于是链接失败，链接器只能抱怨出错。

这里最关键的一点在于：编译器看不到 Stack<T>::Stack()的定义时**并不直接报错，而是做好标记**，这样其它的编译单元如果有这个定义就不会出错了。

总结：可以利用显式实例化的这些性质，隐藏模板的定义源代码，同时限定只能使用特定的几种类型对模板进行实例化！

- 在模板中仍然需要显式的使用 inline 关键字：

  函数模板容易给出一个缺省情况下都是内联的假象。然而这是不正确的，所以如果需要把函数模板实现为内联函数，需要显式的使用 inline 关键字（除非这个函数由于是在类定义的内部进行定义的而已经被隐式内联了）。

- 理解预编译头文件的机制：

  预编译头文件的机理我是明白的，而且也会在 Visual Studio 中使用，但是 GCC 和 Eclipse 中怎么使用预编译头的机制呢？

  - 预编译头文件机制主要依赖于下面的事实：我们可以使用某种方式来组织代码，让多个文件中前面的代码都是相同的。
  - 充分利用预编译头文件的关键之处在于：（尽可能地）确认许多文件开始处的相同代码的最大行数。
  - 绝大多数情况下，可以用一个头文件来包含所有的标准头文件，然后对这个头文件进行预编译。

    C++委员会的很多成员都推崇这种做法，用一个 std.hpp 文件来包含所有的标准头文件，然后其它的代码如果用到了标准库就只要引用这个头文件。

    所以在 VC++中：我应该把所有的标准头文件都放到 stdafx.h 中去，这样就完成了对标准库的预编译。由于 C++项目大了之后编译是很耗时的，这样会节省下相当多的时间，越大的项目越是如此。
  - 管理预编译头文件的一种可取方法是：对预编译头文件进行分层，即根据头文件的使用频率和稳定性来进行分层。（但是这样太麻烦，还是直接包含所有的标准头文件方法比较实用）。
  - 预编译头文件时要注意被宏的影响，宏能完全影响头文件的语义。所以，只预编译标准库是个比较实用而简介的主意。或者绝不使用宏。

- 区分两种约束，语法约束、语义约束：
  - 语法约束：由编译器保证，如 sort 要求必须定义有 operator<；
  - 语义约束：由程序员保证，如 sort 的 operator<的确定义了某种排序规则。

- concept（约束）这个术语通常被用于表示：在模板库中重复需求的约束集合。concpet 可以形成体系，并且可以进一步精华（类似继承）。

  调试模板的主要工作就是判断模板实现和模板定义中的哪些 concept 被违反了。
- 有一个 STLFilt 的实用程序，它提供了一种方法，用于解读多种编译器输出的 STL 错误信息。http://www.bdsoft.com/tools/stlfilt.html
- concepts 是解决调试模板程序困难的最佳和最终解决方案，但是在 C++的新标准(C++11 后面的，11 中 concepts 被否决）出来之前，只能依靠一些其它的方案（通过提前使用参数）。
- 浅式实例化：通过插入没有使用的代码（哑代码）来获取这种实现，这些代码并没有其它的用途，只是在实例化模板代码的高层模板实参不符合底层模板约束时，引发一个错误。
- 使用 Boost Concept Check 库。

第七章：模板术语

- 类类型（class type）包括联合（union），而"类（class）"不包括联合（union）。

  类（class）是指用关键字 class 或 struct 引入的类类型。
- 使用术语"类模板"而不用模板类，意味着这首先是一个类，其次这个类还具有模板的特征。
- 实例化是一个过程，特化是一个实体：

  模板实例化是一个通过使用具体值替换模板实参，从模板产生出普通类、函数、成员函数的**过程**。这个过程最后获得的**实体**（类、函数、成员函数）就是我们通常说的特化。
- **声明和定义**：
  - 声明是引入（或重新引入）一个名称（类名、函数名、变量名）到某个作用域；
  - 定义是**确认构造**（类定义、函数定义）或**分配内存**（变量定义）。
- C++的一处定义原则（ODR）是各种实体的**重新声明**时的约束：
  - 和全局变量与静态数据成员一样，在整个程序中，非内联函数和成员函数只能被定义一次。
  - 类类型（class type）和内联函数在每个翻译单元中最多只能被定义一次，如果存在多个翻译单元，则其所有的定义都必须是等同的。
  - 一个翻译单元（编译单元）是指：预处理一个源文件（一个.cpp 文件就是一个翻译单元）所获得的结果；就是说，它包括#include 指示符（即所包含的头文件）所包含的内容。
- 可链接实体：非内联函数或非内联成员函数、全局变量、静态成员变量，以及从模板产生的上述实体。
- template-id：指模板名称与"紧随其后的尖括号内部的所有实参"的组合，即一个 template-id 标识一个实体。
- 一个基本原则：模板实参必须是一个可以**在编译期确定**的模板实体或者值。这个要求有助于减少

模板实体的运行期开销（满足了这个要求之后，模板产生的类就与普通的类没有任何区别）。

```
template<int N> //这里的N必须是编译期确定的常量
void Print(){
  cout << N << endl;
}
int const i = 9;
int j = 10;
Print<8>();     //OK  8是编译常量
Print<i>();     //OK  i是const，也是常量（编译期可确定它的值）
Print<j>();     //ERROR  j非const，所以非常量（编译期无法确定值）
```

第二部分：深入模板

讨论一些更不常见的问题，也就是当我们深入语言特性，并且希望获得高层次的软件效果时会遇到的一些问题。另外还针对 C++模板的语言特征，预测了 C++模板在将来可能的变化和扩展。

第八章：深入模板基础

- 联合（Union）模板也是允许的（它往往被看作是类模板的一部分）

```
template<typename T>
union AllocChunk {
    T object;
    unsigned char bytes[sizeof(T)];
};
```

- 函数模板声明也可以具有缺省调用实参：

```
template<typename T>
void fill(Array<T> *, T const &=T());
```

注意，当 T 为没有缺省构造函数的类时，只要显式的给 fill 函数提供第 2 个实参，就不会引发错误。即表明了在深层次，编译器只有在真正需要调用 T()时才会去调用 T 的默认构造函数，并不会在一开始就检查 T 是否有默认构造函数。

- **成员函数模板不能被声明为虚函数**，这是一种需要强制执行的限制。因为成员函数模板的使用方式在本质上与虚函数的实现方式是矛盾的。

  成员函数模板的实例化个数，要等到整个程序都翻译完毕后才知道到底有多少个。

  而虚函数实现的机制要求 vtable 的个数必须在编译这个类开始时就完全固定下来。

- 类模板不能和另外一个实体共享一个名称，这一点和 class 不同。

```
int C;
class C;    //可以与 int C 共享名称，没有问题
int X;      //这里已经声明了
template<typename T>
class X;    //错误：'template<class T> struct X'被重新声明为不同意义的符号
```

- 模板名字是具有链接的，但它们不能具有 C 链接（似乎只能是 C++链接），完全可以想象，因为 C 中根本就不支持模板啊。

  并且模板在 C++中通常具有外部链接，除非使用 static 修饰它。

- 现今存在 3 种模板参数：
  - 类型参数

- 非类型参数
- 模板的模板参数

- 在同一对尖括号内部，位于后面的模板参数声明可以引用前面的模板参数名称（但前面的显然不可以引用后面的）
- 在模板声明的内部，类型参数的作用类似于 typedef 名称。

  例如，如果 T 是一个模板参数，就不能使用诸如 class T 等形式的修饰名称，即使 T 是一个要被 class 类型替换的参数也不可以。

```
template<typename ALLOC>
class List{
    class ALLOC *alloc;   //错误： 使用模板类型形参'ALLOC'，在'class'后
    friend class ALLOC;   //错误： 使用模板类型形参'ALLOC'，在'class'后
};
```

  可以想象，这种友元声明的机制非常的有用（如 Singleton 模式模板），在以后可能会被加入，但是 C++11 中依然没有被加入！

  Add：No,这种方式不是应该出现的，因为模板参数 ALLOC 不一定是一个类，他有可能是内置类型如 int，所以当然不应该声明 int 为友元！

- 非类型参数表示的是：在**编译期或链接期**可以确定的常值。必须是下面的一种（其它任何类型都不行）：
  - 整型或枚举类型；
  - 指针类型（包含普通对象的指针、函数指针、指向成员的指针）；
  - 引用类型（指向对象或者指向函数的引有都是允许的）。
- 令人惊讶的是，某些情况下，非模板参数的声明也可以使用关键字 typename

```
template<typename T, typename T::Allocator *alloc>
class List;
```

  第二个 typename 是表示依赖名字 T::Allocator 是一个类型，但它也是一个非类型参数。

- 非类型模板参数的声明和变量的声明很相似，但它们不能具有 static, mutable 等修饰符，只能具有 const, volatile 限定符。但如果这 2 个限定符限定的如果是最外围的参数类型，编译器会忽略它们。
- 非类型模板参数只能是右值：它们不能被取址，也不能被赋值。
- 缺省模板实参：C++98 中只有类模板才能声明缺省模板实参，C++11 中函数模板也可以了。

```
template<typename T=int>
T f(){
      return T();
}
cout << f() << endl;
```

注意：即使类模板的所有的模板参数都具有缺省值，一对尖括号还是不能省略List<> l;

- 函数的实参类型可以通过实参演绎来得到，如果所有的模板实参都可以通过演绎得到，那么函数模板名称后面就不需要指定尖括号。
- 由于函数模板可以被重载，所以对于函数模板而言，显式的提供所有的实参并不足以标识每一个函数（有时提供了所有的实参也只能标识许多函数组成的集合）。

```
template<typename Func, typename T>
void apply(Func f, T x){
  f(x);
}
template<typename T> void multi(T t){
  cout << 1 << ": " << t << endl;
}
template<typename T> void multi(T *t){
  cout << 2 << ": " << *t << endl;
}
int i = 3;
apply(&multi<int>, i);    //这里应该产生二义性
```

在 multi<int>指定了所有的实参之后，所标识的依然有 2 个函数，所以这里的调用就会产生二义性。

在 gcc 4.6.3 下调试时，这段代码能意外的通过编译，但是 gcc 是简单的选取第 1 个函数作为真正调用的函数，即这里的 multi(T t)。把 multi(T t)和 multi(T *t)调换顺序之后，这个调用就直接产生了错误（即 gcc 选取了 multi(T *t)这个函数）。其实这理论上应该是有二义性的，所以这应该是 gcc 的一个错误，直接报二义性会比较好吧。

- SFINAE（替换失败并非错误）原则：SFINAE 是令函数模板可以重载的重要因素，SFINAE 对于 C++模板至关重要。
  允许试图创建无效的类型，但并不允许试图计算无效的表达式。
- 我们平时用的大多数类型都可以被用作模板的类型实参，但有 2 种情况例外：
  - 局部类和局部枚举（指在函数定义内部声明的类型）；
    注意区分局部类和内部类：
    局部类是指在函数定义内声明的类；
    内部类是指在类定义内声明的类。
  - 未命令的 class 类型和枚举类型。
- 非类型实参的编译期整形常值在匹配时会考虑"隐式转换"，这与类型实参不同，即 char 型的常量能匹配 int。
- 模板是编译期的，而多态是执行期的（模板和多态是不相容的）。模板的实参必须是在编译期或链接期就能够确定的值，绝不能是执行期才能知道的值。所以，哪些派生类到基类的隐式转换在

这里都是无效的，对于类型模板参数的隐式类型转换的唯一应用只能是：给实参加上关键字const 或 volatile。

template<typename T, T notype_param>

class C {};

C<Base*, &derived_obj> *error;　　//错误，这里不会考虑派生类到基类的类型转换。

- 有些常值不能作为有效的非类型实参：
  - 空指针常量；
  - 浮点型值；
  - 字符串（可以用 std::string 完美解决）。
- 模板的模板实参必须精确匹配：在匹配过程中，"模板的模板实参"的缺省模板实参将不会被考虑，但是"模板的模板参数"的缺省实参还是有效的（注意区分实参和参数）。

```
template<typename T,
    template<typename ELEM,
    typename ALLOC = std::allocator<ELEM>> //模板的模板实参必须精确匹配
    class CONT>   //这里只能使用关键字 class
class Stack{
public:
  Stack(){}     //必须定义，因为声明变量 s 时构造函数必须被调用
  Stack(Stack<T, CONT> const &);//不必定义，因为"只有哪些被调用的成员会会被实例化
  ~Stack(){}    //必须定义，因为声明变量 s 时析构函数必须被调用
  Stack<T, CONT> operator=(Stack<T, CONT> const &);//不必定义，同上
private:
  CONT<T> elems;
};
Stack<int, vector> stack;
```

这里 vector<T,allocator<T>>的缺省参数就是指："模板的模板实参"的缺省模板（忽略）

而 ALLOC=std::allocator<ELEM>是指：模板的模板参数"的缺省实参（有效）

C++11 中将对这个限制作出改变。

- 模板的模板参数语法上只能用 class 关键字，但是实际上 struct 模板、union 模板都可以当作它的实参。
- 实参的等价性：当每个对应的参数都相等时，就称为这两组模板是相等的。

template<typename T, int I>

class Mix{};

typedef int Integer;

Max<int, 3*3> p1;

Max<Integer, 4+5> p2;

p1,p2 的实际类型是完全一样的，*他们是一个类型的两个变量*。

- 从函数模板产生的函数一定不会等于普通函数，即使这两个函数有相同的类型和名称（想想背后的原理，也许编译器又对函数模板产生的函数使用了 name_mangle 手法）

  （1）从成员函数模板产生的函数永远也不会改写一个虚函数；

  （2）从构造函数模板产生的构造函数一定不会是缺省的拷贝构造函数（所以，三大成员函数都需要 2 个版本 //解决了最早写 qilib 时的一个疑惑）。

- 如果要把一个类模板的实例声明为其它类（或类模板）的友元，该类模板在声明的地方必须可见。然后对于一个普通类就没有这个要求。

```
template<typename T>
class Tree{
  friend class Factory;     //正确，这里是 factory 的首次声明
  friend class Node<T>;  //如果 Node 在此不可见，这条语句就是错的，编译器需要知道 Node 的确是个模板，Node<T>的语法才有效。
};
```

- 在声明友元的同时进行定义（友元插入）必须是首次声明的非受限名称才可以同时进行定义。

首先名称后面不能紧跟一对尖括号，因为如果紧跟了尖括号就表示这是一个模板的实例，那就只能是声明而不能是定义（因为肯定不是首次声明啊）。

然后这个名称必须是非受限的名称，*因为受限的名称一定是对一个已经存在的名称的引用*，而非受限的名称一定不能引用一个模板实例。

当满足了是非受限的名称和后面没有紧跟尖括号之后，由于非受限的名称一定不能引用一个模板实例，并且如果在友元声明的地方还看不到所匹配的非模板函数，那这个友元声明就是函数的首次声明，于是该声明可以定义。

```
void multiply(void*); // 普通函数
template<typename T>  // 函数模板
void multiply(T);
class Comrades {
  friend void multiply(int) {}  // 定义了一个新函数 ::multiply(int)
  friend void ::multiply(void*);// 受限名称，则是引用先前定义的普通函数
  friend void ::multiply(int);  // 引用 template 的一个实体
  friend void ::multiply<double*>(double*);// 带尖括号，一定是一个模板的实例，而且此时编译器必须见到了此 template
 friend void ::error() {}  // ERROR:受限的名称，一定是对已经存在的名称的引用
};
```

- 如果在类模板中定义（非声明）一个友元函数，需要注意下面这个有趣的现象：

```
template <typename T>
class Creator {
  friend void appear()// 定義一個新函式::appear(),但是只有當 Creator 被具現化它才存在
  { }
};
Creator<void> miracle; // ::appear()此时被生成
Creator<double> oops;  // GCC - 错误：'void appear()'已在此定义过
```

于是，apper 函数出现了二次，并且是完全一样的定义，于是违反了 ODR 原则。

因此，一般来说必须在模板内部定义的友元函数类型定义中，包含了类模板的模板参数，这样就不会生成两个完全相同的函数，就不会违反 ODR 了。

```
template<typename T>
class Creator {
  friend void feed(Creator<T>*)  // 不同的 T 會產生不同的::feed 函式定義
  {…}
};
Creator<void> one;// 產生::feed(Creator<void>*)
Creator<double> two;// 產生::feed(Creator<double>*)
```

尽管这些函数是作为模板的一部分生成的，但是这些函数（友元插入的函数）本身仍然是普通函数（而且是内联的），而不是模板的实例。

- 友元模板：让模板的所有实例都成为友元，这就是友元模板

```
class Manager {
  template<typename T>
  friend int ticket() {}  //同样也只能在非受限名称，并没有紧跟尖括号时才能定义
};
```

声明了基本模板之后，所有对应的局部特化和显式特化都自动的成为了友元。

第九章：模板中的名称

- 两个主要的命名概念：
  - **受限名称**：如果一个名称使用了域解析运算符（即::）或成员访问运算符（即.或->）来显式表明它的作用域，那这个名称就是一个受限名称，反之就是非受限名称；
  - **依赖型名称**：如果一个名称（以某种方式）依赖于模板参数，那它就是一个依赖型名称。
- Argument-Dependent Lookup(ADL, 又称 koening 查找)，即依赖于参数的查找：

ADL 只能用于非受限的名称（*因为如果受限了，那就是程序员显式指定的，根本不需要再这么*

费时的去根据参数查找了）。

直观的说，可以认为ADL会查找所有与给定类型（所有的实参）直接相关的所有namespace和class，就好像依次地直接使用这些名字空间进行限定一样。唯一的例外是：它会忽略using指示符。

```
namespace N{
enum E {e1};
void f(E){
	cout << "N::f" << endl;
}
}
void f(int){
	cout << "::f" << endl;
}
::f(N::e1);
f(N::e1);
```

输出如下：

```
::f
N::f
```

看到了，后面的调用甚至优先地选取了ADL查找到的结果。

- 由于类中的友元函数声明可以是该友元函数的首次声明，就牵扯到友元名称插入时的可见性：

  C++标准规定：通常而言，友元声明在外围（类）作用域中是不可见的。

  书上提到了Barton-Nackman方法所依赖的：但是如果友元函数所在的类属于ADL查找关联关集合，那么该友元声明在外围类中是可见的。由于Barton-Nackman方法是解决以前的编译器不支持函数模板重载问题的，所以整个这个已经废弃，现在的GCC已经会对这种使用方法发出警告。

  因此可以这么认为：首次出现的友元声明在外围（类）作用域中是不可见的！

- Maximum munch 扫描原则：C++实现应该让一个标记具有尽可能多的字符。
- 模板中的名称并不能被有效的确定，因为特化的存在（特化可以与原来的定义完全没关系）而使得原来的名称失效，很难确定一个名称到底是不是类型名。

  C++这样解决这个问题：依赖型受限名称并不会代表一个类型，除非在该名称的前面有关键字typename前缀。

  即当类型名称具有以下的性质时，就应该在该名称前面添加typename前缀：

  而且只有当下面的3个条件同时满足叶，才能使用typename前缀（*现代的编译器允许只要是依赖参数的类型就可以使用typename关键字，方便多了！*）

  - 名称出现在一个模板中
  - 名称是受限的

- 名称不是位于指定蕨类继承的列表中，也不是位于引入构造函数的成员初始化列表中
- 名称以来于模板参数

- **显式的 template 提示**：C<T>::template f<int>();

有时候必须在限定符（:: . ->）后面使用关键字 template 来告诉编译器后面的是一个模板。如果**限定符号前面的名称（或表达式）要依赖于某个模板参数，并紧接在限定符后面的是一个 template-id**（就是指一个后面有尖括号内部实参的模板名称）（*这个并不一定是依赖于模板参数的，只需要是一个模板*），那么**就应该使用关键字 template**。

```
template<typename T>
class Shell {
public:
  template<int N>
  class In {
  public:
    template<int M>
    class Deep {
    public:
      virtual void f();
    };
  };
};
template<typename T, int N>
class Weird {
public:
  void case1(typename Shell<T>::template In<N>::template Deep<N>* p) {
    p->template Deep < 6 > ::f(); //抑制 virtual call
  }
  void case2(typename Shell<T>::template In<N>::template Deep<N>& p) {
    p.template Deep < 8 > ::f(); // 同上，且 Deep<8>并不要求依赖于模板参数 N
  }
};
```

因为 C++编译器不会去查找这样的紧跟在依赖于模板参数的名称之后的名称是不是一个模板，所以需要显式的告诉编译器（但编译器为什么这个时候就不去查找这个名称是不是模板呢？）

但是，如果没有必要，不要到处去滥用 template 关键字。

- 使用 using-declaration 的一个漏洞：

```
template<typename T>
class BXT {
public:
 typedef T Mystery;
```

```
 template<typename U>
 struct Magic;
};
```

```
template<typename T>
class DXTT: private BXT<T> {
public:
 using typename BXT<T>::Mystery;//这里必须要有typename标识是一个依赖类型
 using BXT<T>::template Magic;  //这不是C++写法，问题是无法标识Magic是一个模板
 Magic<T> *plink;//错误，编译器不知道Magic是一个模板
};
```

现在这个问题解决了吗？是如何解决的？

在GCC4.6.3下，这个已经完全没有问题了：

```
template<typename T>
class DXTT: private BXT<T> {
public:
 using BXT<T>::Mystery;//不必再写typename了
 using BXT<T>::Magic;  //统一了using-declaration写法
 typename BXT<T>::Mystery m;    //这里使用typename很合情理，BXT<T>是依赖名称
 typename BXT<T>::template Magic<T> *plink;//用::template 显式的表示Magic是一个模板
};
```

更加的统一和直观了：using 引入的是符号，这个符号到底是什么再显式的指出来。

- **一个非常违背直观的查找**：对于模板中的**非依赖型基类**而言，如果在它的派生类中查找一个**非受限名称**，那就会**先查找**这个非依赖型基类，**然后才查找**模板参数列表。

```
template<typename X>
class Base {
public:
 int basefield;
 typedef int T;    //所有，不要去定义这种常用的模板参数名称（尤其是T）的typedef
};
template<typename T>
class D2: public Base<double> {
public:
      void f() {
            basefield = 7;
      }
      T strange;  //永远都是Base<double>::T即int类型
};
```

```
int main() {
	D2<char> d;
	cout << typeid(d.strange).name() << endl;//永远输出 int 类型
}
```

这与直觉完全相反，很容易产生 Bug，应当格外的注意，而且这种情况有一定概率碰到！

即使这种派生是间接的，或者名称是私有的，也是这样查找。

但是，这是为什么呢？完全的违背直觉，而且也肯定不是漏洞，所有的编译器都是这样查找的，深层次的原因是什么呢？

- **名称查找的时机**：C++标准规定如下：
  - 非依赖型名称将会在看到的第一时间进行查找；
  - 依赖型名称将会在实例化的时候才进行查找（*因为特化的存在，依赖型名称在编译期无法确定，必须在实例化时才能确定*）。
- 依赖型基类的名称查找：为了巧妙地解决依赖基类可能被特化的问题，C++标准规定非依赖名称不会在依赖型基类中进行查找（但仍然是在看到的时候马上进行查找）。

  这样就产生了下面的奇怪现象：

```
template<typename T>
class Base{
public:
	int v1;
};
template<typename T>
class Derived : public Base<T>{
public:
	void f(){
		cout << v1 << endl;    //错误：'v1'在此作用域中尚未声明
	}
};
```

解决方案是*使用 this->引入依赖性，这样的话 this->v1 就变成了依赖名称（因为 this 类型是 Derived<T>*），于是它的查找时机也就变成了实例化时，实例化时一切都确定下来了，就可以去依赖基类中进行查找了。*

这就叫做**两阶段查找规则（two-phase lookup）**：在首次看到模板定义时进行第 1 次查找；当实例化模板的时候进行第 2 次查找。

（两阶段查找规则非常重要，在第 10 章会再详细说明）

- 书上说：在查找过程中"非依赖型基类中的名称会隐藏相同名称的模板参数"，这条规则显然是一个疏忽，在将来的标准版本中有可能会修改这个错误。

  在任何情况下，应该尽量不让模板参数名称和非依赖型基类中的名称具有相同的命名！

  ————参见上面的"一个非常违背直观的查找"————

但是：在GCC4.6.3和VC++2010SP1中，这依然存在，为什么呢？C++11是否有对这个疏忽做出修正呢？

第十章：实例化

- 模板的实例化是一个很基础但却有些复杂的概念，因为对于产生自模板的实体（具体类型或函数），它们的定义已经不再局限于源代码中的单一位置。事实上，模板本身的位置、使用模板的位置、定义模板的位置都会对这个（产生自模板的）实体的含义产生一定的影响。
- On-Demand实例化（隐式实例化、自动实例化、按需实例化）：当C++编译器遇到模板特化的使用时，它会利用所给的实参替换对应的模板参数，从而产生该模板的特化实例。
  由于是在遇到模板特化使用时自动产生，所以可以想象在使用模板（特化）的地方一定要让编译器能够访问需要的模板的定义。
- 延迟实例化：编译器只会对确实需要的部分进行实例化，而且它会尽量延迟实例化的时候（*拖到这部分必须被使用的时候*）。
- 延迟的是定义，而不是声明：当编译器实例化类模板时，同时也实例化了该模板的每个成员**声明**，但并没有实例化相应的**定义**，真正的定义会在该成员真正被使用的时候才实例化出来。
  存在两种例外情况：
  - 如果类模板中包含匿名union，那么该union定义的成员同时也被实例化了（*匿名union是一种常用的外围类成员手法*）；
  - 作为实例化类模板的结果，虚函数的定义可能被实例化了，但也可能没有被实例化，这取决于编译器的实例方式。
- “延迟的是定义，而不是声明”的举例：一个类模板一旦被实例化，它的整个类声明就同时实例化了，所以类的声明不能有错（声明有错的话编译器会直接抱怨，但切记仅仅只是声明）；但真正的定义是在这个成员真正被使用的时候才实例化，这个时候就要求定义不能有错了（如果这个成员没有被使用，那么它的定义中的错误会被忽略。但是它的声明一样不能有错）。因为延迟的是定义，而不是声明。

  C++中区分声明和定义太重要了。
- 匿名union：这是一种特殊的技巧，它的成员可以被看成是外围类的成员。匿名成员可以看作是一种构造，用来说明某些类成员共享同一个存储器。
- 当实例化模板的时候，缺省的函数调用实参是分开考虑的。准确而言，只有这个被调用的函数（或成员函数）确实使用了缺省实参，才会实例化该实参。
  这个特性使得我们可以传递没有默认构造函数的类给那些使用了缺省实参的（成员）函数。

```
template<typename T>
void f(T t=T()){...}
```

f(MyClass(6,"OK"); //MyClass 无默认构造函数，但是这样不会实例化 T()，所以调用成功

- 一个重要的原则：在检查涉及模板参数的约束时，编译器会假设该参数"处于最理想的情况"。因为模板的特化和偏特化可能使用一些在普通情况下不适用的参数在该特化下却是有效的，所在编译器只能假设处于最理想的情况之下，延迟到真正的实例化的时候再去做检查。
- 同样，内部类和嵌套类如果没有被使用，也不会立即产生出来，都是会延迟到真正使用的时候。举个例子就一下子明白了：

```
template<typename T>
class Out{
public:
	class In{
	public:
		void f(){ //如果函数的签名（属于声明）出错，编译器会立即抱怨
			//这个 hello 是明显的语法错误，但是编译器并不会去检查。
			//因为是属于定义而不是声明，不被使用的话就不会产生它。
			hello;
		}
	};
	void out_f(){
		cout << "Out f()" << endl;
	}
};
int main(){
	Out<void> o;
	o.out_f();
	return 0;
}
```

输出：Out f()

- 模板实例化是这样的一个过程：根据相应的模板实体，适当地模板参数，从而获得一个***普通**类或者函数*。
- **两阶段查找（two-phase lookup）**：当对模板进行解析的时候，编译器并不能解析依赖名称。于是编译器会在 POI（实例化点）再次查找这些依赖型名称（前面的讲名称查找时详细的分析过）。一点比较特殊的是非受限的依赖型名称会在第一次查找时进行不完整的查找，然后再模板实例化的时候还会再次进行 ADL 查找，详细分析如下：
  - 第 1 阶段发生在解析模板（编译器第 1 次看到模板定义）的时候，这个时候编译器会使用普通查找规则和 ADL 查找规则对**非依赖型名称**进行查找。
    另外，非受限的依赖型函数名称（这里的依赖是指函数的实参是依赖型的）会先使用普通查找规则查找，但只是把查找结果保存起来，并不会试图进行重载解析过程--这是第 2 阶段的查找完成之后才进行的。

- 第 2 阶段发生在 POI 的时候，这时使用普通查找规则和 ADL 来查找**依赖型名称**。

  而非受限的依赖型名称则只进行 ADL 查找（1 阶段进行了普通查找），再把二次查找的结果组合起来进行重载解析。
- ***总结**：第 1 阶段发生在解析模板时，查找非依赖型名称；第 2 阶段发生在 POI 时，查找依赖型名称；同时注意"非受限依赖函数名称"的特殊性。*
- 为什么要引入 2 阶段查找：因为模板展开时的真正语义和模板定义的位置、模板实例化时的位置都是相关的。因此实例化时显然需要引入 2 处不同的声明上下文（在给定位置，所有可以访问的声明所组成的集合）：第 1 处指的是模板定义时的上下文；第 2 处指的是模板实例化时的上下文。
- 实测：在 GCC4.6.3 和 VC++2010 SP1 下编译以下代码：

```
template<typename T>
void f1(T x){
    g1(x);    //x 是依赖名称，它依赖于 T
//  g1(6);    //但如果换成这样写，6 是非依赖的，所以它应该查找不到 g1。gcc 正确报错"错误：'g1'的实参不依赖模板参数，所以'g1'的声明必须可用 [-fpermissive]"，VC++却通过编译并调用了后面的 g1，这是不符合标准的(难道 VC++把所有的名称查找都推迟到了第 2 个阶段吗？)。
}
int main(){
  f1(7);
}
void g1(int){
  cout << "g1(int)" << endl;
}
```

  可以正确的实现 g1 函数名称的查找，按上 2 阶段的查找原则，g1 应该是查找不到编译器出错才对的。*初步估计是 GCC 和 VS2010 SP1 都把内置的类型关联了全局的名字空间，并且把 POI 点放到了编译单元的最末尾处（这是编译器常用的手段），这样的话这段代码可以编译就正常了。*

- POI（实例化点），编译器到底会在哪个位置点把模板实例化出来呢？：
  - 对于**非类型特化的引用**：C++规定它的 POI 定义在"包含这个引用的定义或声明**之后**的最近的名字空间"。
  - 对于产生自模板的**类实例的引用**：C++规定它的 POI 只能定义在"包含这个实例引用的定义或声明**之前**的最近的名字空间域"（因为函数不用确定大小，实质上它只是一个指针；而类对象必须知道它的大小，所以要在之前定义出来）。
  - 对于类模板实例而言，在每个翻译单元中，只有首个 POI 会被保留，而其它的 POI 则被忽略（其实它们并不会被认为是 POI）。

对于非类型而言，所有的 POI 都会被保留。在实际中，编译器会延迟非内联函数模板的实例化，直到翻译单元的末尾处才进行真正的实例化。这样在一个编译单元中也就只有一个非类型的 POI。
所以：在同一个编译单元中多次使用相同的模板特化并不会给编译器带来负担，但是跨编译单元的使用同一个模板特化就会让编译器实例化多次，而最后由链接器抛弃多余的定义。

- 由模板带来的违反 ODR 的问题：由于编译器会在多个编译单元中对模板进行实例化（同一编译单元内由于编译器优化的存在一般不会存在重复定义问题），就会引入违反 ODR 的问题。
  这对类模板没有关系，因为类允许存在重复的相同定义，但是对于函数模板、成员函数模板、静态数据成员就必须解决违反 ODR 的问题。
  常见的解决方案有如下的 3 种（可用的编译器肯定有它的方法解决）：
  - 贪婪实例化：编译器实例化出多个相同的实体，由链接器来抛弃重复的定义。这种改进链接器的方式优势在于保留了源对象之间的原始依赖性，但是这种抛弃的方式是对编译器时间的浪费；（Windows 平台下的编译器基本都是采用贪婪实例化）
  - 询问实例化：维护一个数据库，实体生成之后记录入数据库中，以后的相同实例就不再生成了。但是由于并行编译的存在和与传统编译器模板的冲突，使得这种方式并不常用；
  - 迭代实例化：编译器不实例化任何可链接的特化，链接器在真正需要时再调用编译器实例化出来。这种迭代的方式浪费了很多时间在编译器和链接器来回转移上，而且把错误信息延迟到了链接期。
- 显式实例化：第 6 章讨论过显式实例化，并研究了使用显式实例化来封闭模板定义的源代码，只提供.h 头文件的方式，并且这种使用方式是显式实例化唯一适当、可移植的使用时机。
  由于自动的模板实例化会对编译时间产生极大的负责影响，显式实例化可以压抑编译器的大量无效实例化，因此显式实例化是一个提高编译效率的有效手段。
  但很不幸，C++98 中的显式实例化机制几乎不可用。改进这种方式就变得非常有需要了，于是 C++11 中开始了支持书上提到的扩展，C++11 中叫做"外部模板"。
  extern template void f<int>();　//仅声明不定义，不会引发编译器的实例化
  这种写法很直观，也符合声明 C++中其它外部变量的声明方式。只有在不具备 extern 关键字的情况下，才会引发实例化过程，这样程序员就可以很容易的控制编译器的实例化了（于是可以加快编译速度）。
- C++中，宏并非是能够带来意外强耦合性的唯一构造，导出（export）模型也是带来其它形式的强耦合性，所以导出（export）模型在 C++11 中被抛弃了。

第十一章：模板实参演绎

- 模板实参演绎时匹配实参类型 A 和参数类型 P：

  - 如果被声明的参数是一个引用（即 T&），那么 P 就是所引用的类型（即 T），而 A 仍然是实参的类型。
  - 否则的话，P 就是所声明的类型 T，而 A 是实参的类型。如果 A 的类型是数组或者函数类型的话，还会发生 decay 转型，转化为对应的指针类型，同时还会忽略高层次的 const 和 volatile 限定符。
  - 注意到上面所说的：只有被声明的参数类型是值类型时，才会发生 decay 现象，对于引用参数，绑定到参数的实参是不会进行 decay 的。
- 某些构造不能作为演绎的上下文：
  - 受限的类型名称；如 Q<T>::X 不能被用来演绎模板参数 T
  - 除了非类型参数之外，模板参数还包含其它成分的非类型表达。例如 S<N+1>的类型名称就不能被用来演绎 N。
- 两种特殊的演绎情况（出乎意料啊）：
  - 在取函数模板地址的时候，会进行实参演绎并实例化。

```
template<typename T>
void f(T, T){…}
void (*pf)(char, char) = &f;  //这时会成功演绎并实例化，pf 指向这个实例化的地址
```

  - 与转型运算符模板一起出现的时候：

```
struct S{
  template<typename T, int N> operator T[N]&(){…}
};
void f(int (&)[20]){…}
void g(S s){
  f(s);  //这里也能成功的调用 s 的转型运算符并演绎出正确的模板参数 T、N
}
```

- 模板的实参演绎过程中，当精确匹配不存在时，可以出现一定的（小程度的）宽松匹配：
  - 当 P 是指针（成员指针）和引用类型时，A 可以通过添加限定符（const, volatile）来实现匹配（这很直观，非 const 当然可以匹配 const）。
  - 当演绎过程不涉及到转型运算符时，P 可以是 A 的基类；或当 A 是指针类型时，P 可以是 A 的基类的指针。
- 实参演绎不能应用于类模板和类的**构造函数**，能用应用于普通函数和成员函数。
- 缺省实参不能用于演绎参数类型，即使缺省实参不是依赖型的，也不能用于演绎模板参数。

```
template<typename T>
void f(T x = 42){…}
```

f(); //错误，不能用缺省实参来进行演绎

- Barton-Nackman 技巧（*利用模板实例化的边缘效应生成一个普通函数*）：
  - 首先明白它产生的背景：在以前 C++编译器不支持函数模板重载，并且不支持命名空间；
  - 使用的场景：由于对称性，operator==应该被声明为非成员函数，但是如果编译器不支持函数模板重载，那就只能在一个作用域中声明一个 operator==，显然是不够的。这时 Barton-Nackman 技巧通过把这个运算符作为类的普通友元函数声明在类的内部，解决了这个问题。

```
template<typename T>
struct Array{
    // 前面曾经说过，如果友元插入的定义不依赖于模板参数，则会引起重定义的问题
    friend bool operator==(Array<T> const &a, Array<T> const &b)
    {...}
}
```

  - 为什么 Barton-Nackman 技巧能够正常工作：因为这个被实例化时伴随生成的函数并不是一个模板函数，它只是一个普通的函数，借助于类模板 Array 的实例化的边缘效应被声明了出来，并插入到全局作用域中。由于它只是一个普通函数，所以当然可以重载，规避了编译器不支持函数模板重载。
  - Barton-Nackman 技巧已经不再适用于原来的目的，现在编译器都支持函数模板重载。但该技术很有趣的地方是，它能够在类模板的实例化过程中，伴随生成一个非模板的具体函数（具体函数意思是它的参数是固定的，不需要进行实参演绎）。而且这个函数并不是产生自函数模板，不需要进行实参演绎，但是该函数却属于重载解析规则的作用范围。模板技术真的是具有无限的可能啊！
  - Barton-Nackman 利用了插入式的友元函数，但是在外围作用域中，插入式的友元函数也并不是总是可见的，它只能通过 ADL 才能找到。许多 C++专家认为：友元名称插入很不友好，这会使程序的有效性（某种程序上）依赖于实例化的顺序。
  - *所以：在现代在C++标准和编译器下，Barton-Nackman 技巧几乎没有什么用了。但是这种借助于实例化过程的边缘效应生成一个普通函数还是可以通过编译的，只是它不需要用来解决这种问题了。*

第十二章：特化与重载

- 优先选取特殊的规则：C++语言在其它的条件都一样的情况下，重载解析规则会优先选择更加特殊的模板。

- 语义的透明性：使用特化和重载时，最重要的一点就是保持语义的透明性，不能让模板的使用者对使用不同参数进行实例化时产生不同的效果感到疑惑。
- 不仅是同名模板可以同时存在，它们各自的实例化体也可以同时存在，即使这些实例化体具有相同的参数类型和返回类型。原因在于由于它们不是被同一个模板（即使是重载，也不是同一个）实例化出来的，它们就具有不同的 template-id，进而也就具有不同的签名。
- 只要具有不同的签名，两个函数就可以在同一个程序中同时存在。记住被函数模板生成的函数的签名包含了 template-id，也就包括了它的模板参数和模板实参，一个例子如下：

  ```
  template<typename T1, typename T2>
  void f(T1, T2);
  template<typename T2, typename T1>
  void f(T2, T1);
  ```

  这 2 个从理解上是完全等价的函数模板，实际上会因为不同的 template-id 从而具有不同的签名，就可以同时的在一个程序中出现。

  但是，如果进行 f<char,char>('a','b');类似的调用时，是会产生二义性的，因为这 2 个函数模板的匹配程序完全相同。

  不过，如果这 2 个函数模板的调用出现在同一个程序的不同的编译单元，它们就可以被正确的解析。于是，这 2 个语义上完全相同的模板产生的两个相同的实例化体共存于同一个程序中了。但是，貌似没有什么意义的。
- 正式的排序原则：因为模板的重载解析规则总是更倾向于选取更特殊的规则，如果确定一个模板比另外一个模板更特殊呢？

  如果第 2 个模板针对第 1 份列表可以进行成员的实参演绎（能够进行精确的匹配），而第 1 个模板针对第 2 份列表的实参演绎（精确匹配）以失败告终，那么我们就称第 1 个模板要比第 2 个模板更加特殊。

  这种正式的排序原则通常都能产生符合直观的函数模板选择。然而，该原则偶尔也会产生不符合直观选择的例子。
- 优先选择非模板函数：当其它所有的条件都一样叶，实际的函数调用将会优先选择非模板函数。
- 注意区分显式特化和显式实例化的语法：
  - 显式特化：template<> MyClass<int>{...};　template<> void f(int){...}
  - 显式实例化：template MyClass<int>;　template void f(int);
- 类模板不能被重载，它的透明自定义类模板的能力通常显式特化来实现的。类模板和函数模板都是可以被全局特化的，而且类模板的成员也可以被全局特化。
- 特化和重载模板的区别：函数模板同时支持特化和重载的概念，这 2 个机制似乎提供的功能很相似，但也是存在区别的。

  实际上，全局特化和局部特化都没有引入一个全新的模板或者模板实体，它们只是参原来的泛型模板中已经隐式声明的实例提供另一种定义。在概念上，这是一个相对比较重要的现象，也是特

化区别于重载的关键之处。

- 全局特化的实现并不需要与原来的泛型实现有任何关联，它们只需要在名称上有关联。另外指定的模板实参必须和相应的模板参数列表一一对应，然后如果模板参数具有缺省模板实参，那肥用来替换的模板实参就是可选的。
- 对于特化声明而言，因为它并不是模板声明，所以应该使用普通的成员定义语法，来定义全局类模板特化的成员（也就是说不能使用 template<>前缀）：

```
template<typename T> class S;   //普通模板
template<> class S<char**>{     //全局特化
  void print() const;
}
//下面的定义不能使用 template<>前缀
void S<char**>::print() const {…}
```

为什么是这样的呢？*因为对于一个全局特化来说，它已经是一个普通的类（函数）了，它的定义已经明确了！！！*

- ***小心绝不要同时出现相同的全局特化和实例化**：全局模板特化和由模板生成的实例化版本是不能够共存于同一个程序中的。如果试图在同一个文件中使用这两者，那么通常会导致一个编译期错误。（注意下面的例子，对于函数模板也一样）

```
template <typename T>
class Invalid {};
Invalid<double> x1;                //这里，引发了 Invalid<double>的实例化
template<>class Invalid <double>; //这里再进行 Invalid<double>的全局特化，于是冲突产生了！
```

像这样在一个编译单元中产生这样的错误，编译器会发现并抱怨出来。

但是，*如果这种错误出现在多个编译单元中，就是很难发现*。很多的编译器甚至链接器都会通过类似于这样的在不同的编译单元中的错误。不过这样不但是错误而且是危险的，它不容易被发现且在执行期的行为完全是未知的。

实际中使用模板特化的具体方法：在使用特化时，我们需要特别地小心，并且确认特化的声明对泛型模板的所有用户都是可见的。实际应用中要得到这样的保证，我们应该使用包含模型，并且把模板的特化和模板的声明同样放头文件中特化的声明通常紧跟在模板声明的后面（包含模型再次战胜分离模型的地方）。

不要轻易去特化外部泛型资源：如果泛型资源来自于外部资源（诸如不能被修改的头文件），解决的办法是创建一个包含特化的头文件，并让特化声明位于泛型模板之后。这样在任何需要使用这个泛型资源的地方就包含我们创建的新模板，就可以避免这种"难以发现"的错误。不过，如果不是确实具有特殊的用途和目的的话，我们通常都避免特化来自于外部的泛型资源。

- 全局函数模板特化与全局类模板特化非常类似，除了引入了重载和实参演绎这 2 个概念。
- 全局函数模板特化不能包含缺省的实参值。然后对于（即要被特化的）模板所指定的任何缺省实

参，<u>显式特化版本都可以应用这些缺省实参</u>。

- 全局特化的结果并不是一个模板，它只是一个普通的类或者函数。
- 由于全局特化的结果只是一个普通的对象，这样函数模板（类模板没问题）的特化就**有可能产生重定义**。

  ```
  template<typename T> void f(T) {…}
  template<> void f(int){…}        //特化是个普通函数，普通函数不应该在头文件中定义！
  ```

  由于前面提到的特化必须紧跟在模板声明后面，于是如果 a.cpp, b.cpp 都包含了这个头文件，那么 template<>void f(int)就重定义了。解决方案有 2 个：

  - 把这个特化声明为内联函数（内联函数可以在头文件中定义）；
  - 仅仅把这个特化的声明紧跟在模板后面，这个特化的定义放到源文件中去。
- 在使用全局成员特化之后，类的其它成员仍然将默认地产生自原来的模板。
- 就像前面说的，**我们必须把特化紧跟在模板的后面，但是为了避免重定义，我们又只能把特化的声明紧跟在模板的后面**。

  于是对于全局成员特化，就必须有一种成员声明而不定义的语法。<u>尽管对于普通类的成员函数和静态成员变量而言，非定义的类外声明是不允许的；但是如果是针对类模板的特化成员，该声明是合法的</u>（*书上说的是对的，的确是这样的！*）。

  eg 如下：

  ```
  //template.h
      template<typename T>
      class Test{
          static T value;
      }
      //类模板特化的非定义类外声明而非定义（如果这里是定义，那么就又可能出现重定义）
      template<> int Test<int>::value;
  //main.cc
      template<typename T> T test2<T>::value = T();   //普通的类外定义
      template<> int Test<int>::value = 6;             //类模板特化的类外定义
      int main() {
          cout << Test<int>::value << endl;
      }
  ```

  输出：6

  注：GCC 中可以正确的处理类模板特化的非定义类外声明，正确的输出了 6！
- 局部的类模板特化：一种常见的应用就是针对指针类型和非指针类型进行不同的特化。

  ```
  template<typename T>
  ```

class MyClass{...};

template<typename T>

class MyClass<T*>{...};

- 当进行匹配的时候，全局特化会优于局部特化（更特殊嘛）。
- 对于局部特化声明的参数列表和实参列表，存在以下的一些约束：
  - 局部特化的实参必须和基本模板的相应参数在各类上是匹配的；
  - 局部特化的参数列表不能有缺省实参，但是可以使用基本模板的缺省实参；
  - 局部特化的非类型实参只能是非类型值，或者是普通的非类型模板实参，而不能是更复杂的依赖型表达式（如 2*N，其中 N 是模板参数）；
  - 局部特化的模板实参列表不能和基本模板的参数列表完全相同。
  - 示例如下：

```
template<typename T, int I = 3>
class S;                    //基本模板
template<typename T>
class S<int, T>;            //错误：参数类型不匹配
template<typename T = int>
class S<T, 10>;             //错误：不能具有缺省参数
template<int I>
class S<int, I*2>;          //错误：不能具有非类型表达式
template<typename U, int K>
class S<U, K>               //错误：局部特化和基本模板之间没有本质区别
```

- **最特殊规则**：如果能够找到多个匹配，那么将会选取"最特殊"的特化（和重载函数模板所定义的原则一样）；如果未能找到"最特殊"的一个特化，即存在几个特殊程序相同的特化，那么程序将会包含一个二义性错误。
- 类模板局部特化的参数个数是可以和基本模板不一样的：即可以比基本模板多，也可以比基本模板少。
- 借助于模板特化可以避免出现无限递归的模板定义（模板元编程的基础），Erwin Unruh 可能是第一个发现这种能力可以带来有趣的 template metaprogramming 的人。模板元编程是指借助于模板实例化机制，在编译期执行一些重要的计算。
- 重载函数模板和函数模板的局部特化看起来是类似的，它也存在一些细微的区别：
  对于特化，编译器在看到一个调用时，只会查找基本模板；而特化则是在后来需要决定调用哪一个具体的实体时才会被考虑的。相反，对重载函数模板进行查找的时候，所有的重载函数模板都必须被放入重载集合里面，再根据重载规则从这个集合中选取。而且这些重载函数模板还可以来自不同的名字空间和类，这将会增加无意中重载某个模板名称的可能性。

第十三章：未来的方向

- 尖括号 Hack：连续的 2 个>，即>>会被编译器默认处理成右移符号，这很不直观，而且编译器解决这个问题是很容易的。

  注意到修复尖括号 Hack 还存在一些复杂的边缘问题：

  strange<Buf<16>>2> >(); 其中的 16>>2 就是想要做一次右移操作，如果要修复这个 Hack，就应该能处理这样的情况，解决方案是使用括号 starnge<Buf<(16>>2)>>();

  还有一个问题就是<:实际上等价于一个方括号[。于是 vector<::C>的代码在编译器看起来会是 vector[:C>，这显然是一个错误。

  C++11 现状：已经解决了尖括号 Hack，现在的>>会默认为模板的结束符；

  <::的问题在 GCC 中依然需要在前面添加一个空格，但是 VC++2010 没问题。

- 放松 typename 原则：typename 原则在 C++98/03 中太严格了，以至于非常的不直观。

```
template<typename T>
class Array{
public:
 typename T ElementT;
};
template<typename T>
void clear(typename Array<T>::ElementT &p);       //1：正确
template<>
void clear(typename Array<int>::ElementT &p);     //2：错误
```

  第 1 个 typename 是必须的，而第 2 个 typename 又一定不能有。

  产生这样奇怪的现象的原因：是由于 Array<T>::ElementT 是依赖名称，Array<T>的具体定义在 clear 函数声明时是无法确认的，所以必须显式的告诉编译器 ElementT 是一个 typename。而对于 Array<int>::ElementT 是非依赖名称，它的定义在进行这个声明时就已经可以被确定下来了，所以不需要手动的去告诉编译器 ElementT 是一个 typename，编译器自己知道。

  但是这还是非常的不直观的，因为很多程序员分不清楚依赖名称和非依赖名称。比较好的方法就是只要 ElementT 是一个类型名称，都可以在前面添加 typename，由编译器忽略掉不必要的 typename 就好了。

  C++11 现状：GCC 中已经可以接受多余的 typename，VC++2010 也已经可以接受，这个问题已经不再是问题了。

- 缺省的函数模板实参：类模板支持缺省的实参，而函数模板却不支持，这显然是不符合正交性的，函数模板支持缺省的实参也不再是技术问题，*C++11 中也明确支持了函数模板的缺省实参。*

  就像普通的缺省实参一样，如果某个参数具有缺省实参，那么后面的每个参数都必须具有缺省模板实参。然而对于函数模板而言，可以通过演绎来推导出后面的这些参数，这是函数模板由于实

参演绎所带来的更强的特性。

- 字符串文字和浮点型模板实参：
    - 字符串文字作为模板实参时会与直觉不同。看起来相同的 2 个字符串却不一定（看编译器实现）具有相同的地址，由于字符串文字默认为 const char*类，所以 MyClass<”X”>与 MyClass<”X”>由于地址不同就成了完全不同的 2 个类型。

      我觉得语言可以不必要做这样的扩展吧，使用 std::string 就完全的解决了问题。
    - 浮点数作为非类型参数也不被支持，但是支持它们并没有技术困难。不过，我觉得可能由于浮点数天生存在的误差以及一定程度上的不可比较性，标准禁止浮点数作为非类型参数也是有道理的。
    - C++11 中对这 2 条都没有改变，而且我觉得也的确不应该改变，改变了反而容易引起误会。
- 放松模板的模板参数的匹配：用于替换模板的模板参数的模板必须能够和模板参数的参数列表精确匹配，连缺省实参也不允许使用。这有些不符合直觉，所以以后的版本可能放松这个限制。
- typedef 模板：目前的标准只能 typedef 模板的一种特化，而不能 typedef 模板，区别如下：

  typedef vector<vector<int>> Vec2DInt; //OK

  template<typename T>

  typedef vector<vector<T>> Vec2D; //ERROR，但是这样显然更有用 Vec2D<int> v;

  C++11 已经提供了using 的新语法来支持这个扩展：

  templaet<typename T>

  using Vec2D = vector<vector<T>>;
- 函数模板的局部特化：类模板允许局部特化而函数模板不支持，这也是不符合正交性的。特化和局部特化机制的威力还是比重载更强大一些，因为未来有可能提供对函数模板的局部特化机制的支持。

  不过，某些语言的设计者担心把局部特化和函数模板重载交互使用，将会出现问题。

  在C++11 中，依然不支持函数的局部特化。我觉得可能是因为这可以很方便的由函数对象来实现， 利用类的偏特经来模拟，的确没有必要再引入这样的复杂性了。
- typeof 运算符：很多时候需要根据实参的类型来进行另外的类型推导，最简单的就是+法模板：

  template<typename T1, typename T2>

  Array<???*(typeof(x+y))*> operator+(Array<T1> const &x, Array<T2> const &y);

  如果推导出+法结果的类型，这需要一种类型推导的机制。不止需要一种类型推导机制，还需要一种新的函数声明语法，在参数后面声明返回值的类型。因为???的类型必须在 x 和 y 出现之后再进行推导，它肯定不能在编译器遇到 x,y 符号之前就进行推导。

  而且，typeof 必须是一个编译期的运算符，它不会考虑运行期的多态特性。

  C++中提供了完全的支持：一个新的decltype 运算符和新的函数声明语法

  template<typename T1, typename T2>

auto operator+(Array<T1> const &x, Array<T2> const &y) -> decltype(x+1)

{...}

- 命名模板实参：对模板参数进行命名，就可以依靠参数名字进行实例化而不是仅仅依靠参数的顺序。

  template<typename T,

  Move: typename M = defaultMove<T>,

  Copy: typename C = defaultCopy<T>>

  Class MyClass{...};

  MyClass<int, Copy: anotherCopy> var;

  感觉上这样的意义不大，这种特性估计不会被支持，C++不应该再引入没必要的复杂性了。

- 静态属性：在运行期进行类型识别是使用 RTTI，但是在编译期根据不同的静态属性选择不同的模板行为是依靠 trait 技术来实现的，比较麻烦。而且这种特性又是非常常用的，所以应该由语言来提供这样的扩展，在编译期根据类型不同的静态属性来决定不同的行为。

  std::type<int>::is_bit_copyable;

  std::type<int>::is_union;

  C++11 中也提供了这个支持，叫做 Type traits：

  std::is_integral<T>::value;

  std::is_floating_point<T>::value

  可以想象，以后语言还可以在静态属性的基础上实现反射的机制。

- 客户端的实例化诊断信息：许多模板都会对它们的参数强加一些**隐式的**要求，当该模板的实例化的实参不能符合这些要求时，就会产生一个泛型错误。但是模板的错误信息就像天书，现在几乎是不可读的，所以优化模板的错误诊断信息变的非常必要。

  有很多种建议来解决这个问题，如其于上面一条讨论的静态属性的方法、哑代码的方法等。

  concept 是标准委员会用来解决这个问题的方案，C++11 在对 concept 进行了巨大的工作之后，由于 concept 的复杂性还是把 concept 给否决了，但是以后应该会被加入到语言标准中来。

- 重载类模板：模仿函数的重载，基于模板参数之间的差异对类模板进行重载是完全可能的。

  但是，我觉得这种机制的必要性似乎不是很强烈吧。

  在语言不支持类模板的重载和变长模板参数时，boost::tuple 是这样实现的：

  template<typename T0 = null_type,

  typename T1 = null_type,

  ...

  typename T9 = null_type>

  class tuple{...};

  很明显，boost::tuple 这样的实现方式效率不高。如果支持类模板的重载就好多了，当然变长模

板参数就更好了，不止效率好而且支持无限多个类型参数（boost::tuple 最多 10 个）。

- list 模板参数（变长模板参数）：使模板参数支持变长参数之后可以实现声明一个参数个数不固定的函数或者定义一种具有成员个数不固定的类型结构。

  ```
  template<typename T, ... list>
  ```

  一个必要的规则：让重载和特化规则优先选取不具有 list 参数的模板。

  使用这个规则，就可以利用 list 模板参数实现很多有趣强大的 metaprogramming：

  ```
  tempalte<typename T>
  struct ListProps{
     enum { length = 1 };
  }
  template<... list>
  struct ListProps{
     enum { length = 1 + ListProps<list[1...]>::length };
  }
  ```

  其中 list[1...]是选取 list 模板参数中的子集，这有点类似于动态语言如 ruby 等的语法了。

- 布局控制：模板编译中经常需要得到一个类型的 aligment requirement，这与 sizeof 略有不同。现在的许多 C++编译器已经支持一个名为__alignof__的运算符，来返回相应的 aligment requirement，将来可能引入一个关键字 alignof 来实现。

  C++11 中已经引入关键字 alignas 来实现这个功能。

- 初始化器的演绎：
  - 初始化之后的类型实际上可以被编译器推导出来的，不必须声明一些冗长的类型名。

    C++11 中使用 auto 关键字进行类型推导：auto iter = v.begin();
  - 函数模板和成员函数模板都可以通过实参演绎来推导实参类型，但是构造函数却不可以。一种可能的变化是使得构造函数实参来演绎模板实参，但是由于存在重载的构造函数（包含构造函数模板），这种演绎会变得很复杂以致很难给出准确的定义。

- 函数表达式：表达式模板技术能够被准确地用于创建小的仿函数，而且不会涉及到显式声明的开销。lambda 表达式用于声明一些小的函数是非常方便的，同样的解决方案在其它语言中有时叫做闭包，或者类似于 Java 中的匿名内联类。

  C++11 中已经定义了完全的 lambda 表达式的语法，非常好用。

- 最后总结一下：C++11 还是解决了作者提出的绝大部分问题和改进的，可见 C++11 的完善和改进是非常大的，标准委员会一定付出了巨大的努力和大量的工作。

第三部分：模板与设计

模板的不同之处在于：它允许我们在代码中对类型和函数进行参数化。
把“局部特化”和“递归实例化”组合起来，将会产生出人意料的强大威力。这种静态的递归能力，已经被证明是图灵完备的。

第十四章：模板的多态威力

- 动多态与静多态
  - 动多态：通过继承和虚函数实现的多态是动多态，是绑定的和动态的：
    - 动态的意味着是在**运行期**完成的。
    - 绑定的意味着公共行为明确地依赖于基类的接口。
  - 静多态：通过模板实现的多态是静多态，是非绑定的和静态的：
    - 静态的意味着是在**编译期**完成的。
    - 非绑定的意味着参与多态的类型并不存在明确的接口，而是存在一种隐式的公共性。
- 动多态最引人注目的特性或许是处理异类容器的能力。
  静多态不能透明的处理异类的集合，这也是静多态的静态特性所强加的约束：所有的类型都必须能够在编译期确定（静多态是编译期多态、动多态是运行期多态）。但是使用静多态时集合类型就不再局限于指针，从而能够在性能和类型安全方面带来显著的好处。
- 动多态与静多态的优缺点分析
  - 动多态：
    - 能够优雅的处理异类集合；
    - 执行代码比较小；
    - 可以进行完全编译，不需要发布源代码。
  - 静多态：
    - 不需要公共基类，就可以与不能修改的类型结合使用（如标准库类型和不能修改的库）；
    - 所成的代码效率高；
    - 对于只提供部分接口的具体类型，如果在程序中只使用到了这一部分接口，那么也可以使用该类型，不必在乎该类型是否提供了其它部分的接口；
    - 更好的类型安全性。
  - 奇异递归模板式是对这两种多态的组合使用。
- 在C++的上下文中，我们有时也把泛型程序设计定义为运用模板的程序设计。到目前为止，在

C++泛型程序设计领域中，最显著的贡献就是STL。

- STL中泛型程序设计的“粘合剂”就是：由容器提供的并且能被算法使用的迭代器。迭代器之所以能够肩负这样的任务，是由于容器为迭代器提供了一些特定的接口，而算法所使用的正是这些接口。我们通常把每个这样的接口称为一个 **concept（即约束）**，它说明一个模板（即容器）如果要并入STL框架就必须履行或实现这些约束。
- 容器类型是当初把模板引入C++程序设计语言的主要动力。
- C++中静多态的一个显著的优点就是可以与哪些不能修改的类型协作多态的能力。静多态不需要明确的约束，不需要像动多态哪样继承自明确的基类。

  鸭子类型：*“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”*

  C++中的模板就是采用了鸭子类型的思想，只要你提供了这些约束，就可以被静多态使用，而不必去显式声明它。而那些只有动多态的程序设计语言如Java,C#等，都必须使用庞大的多态类型来对内建类型进行包装（如C#就由此引入了装箱、拆箱的概念），这很明显会导致效率的降低和内存使用量的大幅增加。

第十五章：trait与policy类

- Policy类和trait（或称为trait模板）是两种C++程序设计机制，它们有助于对某些额外参数的管理，这里的额外参数是指：在具有工业强度的设计中所出现的参数。
- 对于内置类型和一些不能被修改的类型（如来自不能修改的库），如果我们需要获取它们的与类型相关的额外信息，特化往往是个最好的选择。如果没有特化，对于这些类型获取额外的信息就变的非常困难，甚至无能为力。

```
template<typename T>
class AccumulationTraits;
template<>
class AccumulationTraits<int> {
public:
  typedef long AccT;      //获取了与内置类型int相关的类型信息(它的提升类型)
}
```

- 在类的内部，C++只允许我们对整型和枚举型初始化成静态成员变量。

```
class MyClass {
  static int const zero1 = 0;  //OK,允许static const的整型和枚举型在类定义内部初始化
  static double const zero2 = 0.0;  //ERROR, 不允许double类型也这么做
```

}

一个简单直接的解决方案就是在类定义的外部进行初始化

```
class MyClass {
  static double const zero2;
}
double const MyClass::zero2 = 0.0;
```

这种方法是可行的，但是由于把定义和声明分开了，会影响编译器的优化，一个更好的解决方案是使用静态的函数（用静态函数返回值就没有任何类型上的限制）：

```
class MyClass {
  static double zero2() { return 0.0; }
}
```

多数 C++编译器都能对这种情况进行内联优化，于是能获得更高的效率。

- 三种类型的 traits
  - ```
    class AccumulationTraits<int> {
    public:
      typedef long AccT;            //fixed traits
    }
    ```
  - ```
    class AccumulationTraits<int> {
    public:
      static int const zero = 0;        //value traits
    }
    ```
  - ```
    template<typename T,
               typename AT = AccumulationTraits<T> >     //参数化 traits
    class Accu { ... }
    ```
- 一个 policy 类就是一个提供了一个接口的类，该接口能够在算法中应用一个或多个 policy。有点类似于 Strategy 模式，提供一个类来完成一项功能。
- Trait 和 policy 的区别：
  - trait：用来刻划一个事物的（与众不同的）特性；
    - trait 更加侧重于类型的特性，表述了模板参数的一些额外属性
  - policy：为了某种有益或有利的目的而采用的一系列动作；
    - policy 理加侧重于行为，表述了泛型函数和泛型类的一些可配置的行为。
  - 这两个概念之间存在着很多的共同点，它们之间只是一条很模糊的界限，也还存在一些交叉的地方。
- trait 和 policy 通常都不能完全代替多个模板参数；然而，trait 和 policy 确实可以减少模板参数的个数，并把个数限制在可控制的范围以内。
- 一种简单的参数排序策略就是根据缺省值使用频率递增地对各个参数进行排序。这样就会最大限度的利用缺省参数。
    ```

- **类型函数**：类型函数是个全新的概念，它接收某些类型作为实参，并且返回该类型的相关信息或者生成一个类型作为返回结果。

  sizeof 就是一个非常有用的、内建的类型函数，它返回一个描述给定类型实参大小（以字节为单位）的常量。

  比如上面使用的 AccumulationTraits 类，它接受一个类型作为它的模板实参，通过 AccumulationTraits<T>::AccuT 返回这个类型的提升类型，这就是一个典型的类型函数。
- trait 的实现可以被看成是对现在类型的一种扩展。因此，即使是基本类型或位于封闭程序库中的许多类型，都可以通过 trait 手法为这些类型定义类型函数。
- 经常会只使用一个 trait 来表示一个类型的许多信息甚至全部类型相关的信息，如标准库中的 iterator_traits 就包含了与某一类型迭代器相关的大量信息。

```
template<typename T>
struct iterator_traits<T*> {
  typedef  T                            value_type;
  typedef  ptrdeff_t                    difference_type;
  typedef  random_access_iterator_tag   iterator_category;
  typedef  T*                           pointer;
  typedef T&                            reference;
}
```

- 判断是否为 class 类型(gcc 能正确编译，VC++2010 不符合标准不支持)

```
template<typename T>
class IsClassT{
private:
  typedef char One;
  typedef struct {char a[2];} Two;
  template<typename C> static One test(int C::*);
  template<typename C> static Two test(...);
public:
  enum {Yes = sizeof(IsClassT<T>::test<T>(0)) == 1};
  enum {No = !Yes};
};
cout << IsClassT<int>::Yes << endl;
cout << IsClassT<MyClass>::Yes << endl;
```

  这个强大的模板利用了以下知识：
  - SFINAE 原则（替换失败并非错误）：当某一个 test 匹配失败时，并不会引发编译错误，而是进行另外一个重载的匹配，只要有任何一个 test 重载能够匹配成功，这就是一个正确的使用。

- sizeof 运算符并不会引发实例化（typeid 也不会），因为实际上并没有真正的调用该函数。所以 test 静态函数都只需要声明而不需要定义。
- ...变长参数具有最低的优先级：匹配会优先考虑 int C::*的版本，只有在这个版本匹配失败时才会去尝试...变长参数的版本。
- ...变长参数会能够与任意类型的实参匹配，所以保证了这个模板不会引发编译期错误。

- 指向 void 的引用是不合法的。
- 一个很强大的 IF—THEN—ELSE 选择分支模板：

```
template<bool C, typename Ta, typename Tb>
class IfThenElse;                      //普通 IfThenElse 模板
template<typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb> {      //针对 true 时的特化
public:
  typedef Ta ResultT;
};
template<typename Ta, typename Tb>
class IfThenElse<false, Ta, Tb> {      //针对 false 时的特化
public:
  typedef Tb ResultT;
};
```

特化机制真的很重要，通过特化轻松的实现了模板的分支选择。

- IF_THEN_ELSE 模板的一个很好的使用例子，选择两种类型的提升类型的默认模板：

```
template<typename T1, typename T2>
class Promotion {
public:
   typedef  typename IfThenElse<(sizeof(T1)>sizeof(T2)>,
                                T1,
                                T2>::ResultT  ResultT;
};
```

这样就能实现自动的选择容量更大（占字节数更多）的类型作为提升类型，这对大多数情况下都是适用的。

- 如果模板参数只是出现在函数参数的限定符里面，那么就不能使用实参演绎来调用该函数。如：

```
template<typename T1, typename T2>
void foo(typename RParam<T1>::Type p1,
         typename RParam<T2>::Type p2>
{ ... }
```

解决办法是使用一个内联的包装函数模板，像 make_pair 一样：

```
template<typename T1, typename T2>
inline void foo_wapper(T1 const &p1, T2 const &p2)
{  foo<T1, T2>(p1, p2);  }
```

使用内联函数期望编译器能够把这层间接性带来的效率损失消除掉，实际上编译器也可以做到。

- 通常而言，客户端的代码都不会涉及 trait：因为缺省的 trait 类都可以满足一般的需求，而这些 trait 类都是一些缺省的模板实参，因此在客户端代码中并不需要出现。
- trait 的最初目的是为了减少将要模板实参的数量，而如果我们在模板参数中仅仅封装一个 trait 的话，那么将达不到最初的目的，所以一般会封装与主类型相关的多种信息。
- 如果在 trait 里封装了与行为相关的 trait 时，就可以称之为 policy trait.

第十六章：模板与继承

- 通过模板与继承的结合，借助所谓的参数化继承，能够产生出精彩的技术火花。
- **命名模板参数技术**，代码如下所示：

```
#include <iostream>
#include <typeinfo>
using namespace std;

class DefaultPolicy1{};
class DefaultPolicy2{};
class DefaultPolicy3{};
class DefaultPolicy4{};

class DefaultPolicies{
public:
  typedef DefaultPolicy1 P1;
  typedef DefaultPolicy2 P2;
  typedef DefaultPolicy3 P3;
  typedef DefaultPolicy4 P4;
};

class DefaultPolicyArgs : virtual public DefaultPolicies{};
template<typename Policy>
class Policy1_is : virtual public DefaultPolicies{
public:
  typedef Policy P1;
};
template<typename Policy>
class Policy2_is : virtual public DefaultPolicies{
public:
  typedef Policy P2;
};
```

```
template<typename Policy>
class Policy3_is : virtual public DefaultPolicies{
public:
  typedef Policy P3;
};
template<typename Policy>
class Policy4_is : virtual public DefaultPolicies{
public:
  typedef Policy P4;
};

template<typename Base, int D>
class CanMultiDerivedFromOneClass : public Base{};

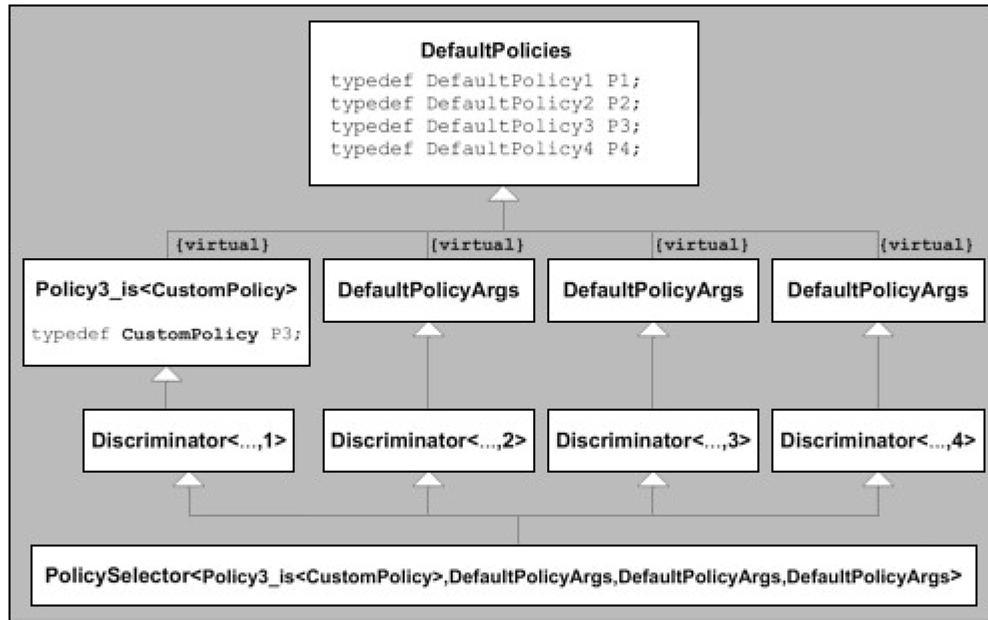
template<typename Setter1, typename Setter2, typename Setter3, typename
Setter4>
class PolicySelector : public CanMultiDerivedFromOneClass<Setter1, 1>,
                    public CanMultiDerivedFromOneClass<Setter2, 2>,
                    public CanMultiDerivedFromOneClass<Setter3, 3>,
                    public CanMultiDerivedFromOneClass<Setter4, 4>
{};

template<typename PolicySet1 = DefaultPolicyArgs,
        typename PolicySet2 = DefaultPolicyArgs,
        typename PolicySet3 = DefaultPolicyArgs,
        typename PolicySet4 = DefaultPolicyArgs>
class MyClass{
public:
  typedef PolicySelector<PolicySet1, PolicySet2, PolicySet3, PolicySet4> Policies;
};

int main(){
  typedef MyClass<Policy2_is<int>, Policy4_is<double> > MyClassDef;
  MyClassDef::Policies::P1 p1;
  MyClassDef::Policies::P2 p2;
  MyClassDef::Policies::P3 p3;
  MyClassDef::Policies::P4 p4;
  cout << typeid(p1).name() << endl;
  cout << typeid(p2).name() << endl;
  cout << typeid(p3).name() << endl;
  cout << typeid(p4).name() << endl;

  return 0;
}
```

代码的类图如下：



这些代码中有以下需要注意的几点：

- CanMultiDerivedFromOneClass<T, N>所引入的中间层是也是一个经常使用的技术。当使用多继承时 class MyClass : public T1, public T2，并且这个基类还依赖于模板参数，那么就有可能使得这 2 个基类是同一个类型。那么，就必须使用中间层 CanMultiDerivedFromOneClass 了，即使 T1 和 T2 是同一个类型，也由于后面的非类型模板参数 N 不同变为不同的类型，于是消除了这个编译错误。
- PolicySelector 使用了参数化继承技术，它继承了自己的模板参数类型。
- 由于最后可能会有多个 Policy*_is 模板类和 DefaultPolicyArgs 继承自 DefaultPolicies, 所以这里使用了 virtual 继承。
- 派生类 Policy*_is<>重定义了 P*，根据优势规则，重定义的类型隐藏基类中的定义，所以这里不会存在二义性的问题。
- Policy*_is 和 DefaultPolicyArgs 类实际上永远都不会被实例化（只用到了它们的 typedef），所以它们实际上不会消耗任何内存和影响性能。

- C++中的类常常为"空"（作为接口等等情况），但标准又规定了不能存在"0 大小"的类，但是这扇门也没有彻底关死。C++标准规定，**当空类作为基类时，只要不会与同一类型的另一个对象或子对象分配在同一地址，就不需要为其分配任何空间**。做到了真正的 0 大小。
标准作这些规定和留这些后门的根本原因在于：**我们必须能够比较两个指针是否指向同一对象**。

由于标准留下的这扇门，就产生了空基类优化技术。

- **空基类优化**：在模板中考虑这个问题特别有意义，因为模板参数常常可能就是空类（大部分 trait 就是空类）。

template<typename T1, typename T2>

```
class MyClass {
   T1 a;
   T2 b;
}
```

T1，T2 很可能是空类，那么这样老老实实的表示 MyClass<T1,T2>就不能得到最优内存布局，每个这样的实例可能会浪费一个字的内存。把模板参数直接作为基类就可以解决：

```
template<typename T1, typename T2>
class MyClass : private T1, private T2 {…};
```

但是这种方式引入的复杂性会带来不必要的麻烦。一个更加常用的方式是：如果一个模板参数类型必须为类，该模板的另一个成员类型不是空类，那么就借助"空基类优化"的东风，把可能为空的类型参数与这个成员合起来。

```
template <typename CustomClass>
class Optimizable {
  private:
    CustomClass info;     // might be empty
    void*       storage;
    …
};
```

将其改写为：

```
template <typename CustomClass>
class Optimizable {
  private:
    BaseMemberPair<CustomClass, void*> info_and_storage;
    …
};
```

其中 BaseMemberPair 模板的声明如下：

```
template <typename Base, typename Member>
class BaseMemberPair : private Base {  …  };
```

这样就把这样的复杂性处理交给了BaseMemberPair 模板去，而不是在每个需要进行这样优化的地方去关心和小心处理把成员变成基类可能引起的问题。

boost 库中包含了一个更好、更精致的模板 compressed-pair，可以完全取代这里的 BaseMemberPair（这样优化起来非常方便！）。

这样，就可以利用空基类优化得到更好的内存布局，虽然增加了复杂性，但是对于许多库来说性能能够得到显著的提供，一定的复杂性是值得的。

- 奇特的递归模板模式（CRTP）：派生类将本身作为模板参数传递给基类的技术。
  这种技术的 2 个最常见的应用就是实现 Singleton 模板和引用计数器 Count 模板。
  CRTP 类似于参数化继承，但是 CRTP 并不等价于参数化继承。
- C++允许通过模板直接参数化 3 种实体：类型、常数、模板。同时，模板还能间接参数化其它属性，通过把模板参数作为自己的基类，还可以定制某个函数是否为虚拟性，这很容易想到。

第十七章：metaprogram

- metaprogramming 含有"对一个程序进行编程"的意思。换句话说，编程系统将会执行我们所写的代码，来生成新的代码，而这些新代码才真正实现了我们所期望的功能。
- metaprogramming 最大的特点在于：某些用户定义的计算可以在程序翻译期（通过模板实例化）进行。
  而这通常都能够在性能或接口简单性方面带来好处，甚至为两方面同时带来好处。
- 模板实例化机制是一种基本的递归语言机制，可以用于在编译期执行复杂的计算。因此，这种随着模板实例化所出现的编译期计算通常就被称为 template metaprogramming（模板元编程）。实际上 template metaprogramming 后面所做的工作就是递归的模板实例化。
- 枚举值和静态常量的选择：
  - 模板元编程使用的必须是在编译期就能够确定的值，也就是枚举值或静态常量值。
  - 枚举值和静态常量的不同之处：静态成员变量只能是一个"左值"；而枚举值却不是一个"左值"（*即它没有地址*）。
    因此，枚举值也就相当于"字面值"的形式，当通过引用传递这个完成计算的值时，编译器不会为它分配任何内存；但如果使用静态成员常量时，会强制编译器实例化静态成员的定义并为该定义分配内存，这就引入的额外的效果。
    所以，一般来说，对于整型数，枚举值优先于静态常量。
- 模板元编程中引发的实例化实体过多的问题：模板元编程中的实例化实体数会随着递归的深入急剧的膨胀。原因之一可能就是分支的结构，因为分支结果是递归编程中最常用的结构，例如：
  result = (N<mid*mid) ? Sqrt<N, LO, mid-1>::result : Sqrt<N, mid, HI>::result;
  根据(N<mid*mid)的结果选择一条分支进行，但是编译器却会实例化所有分支上的 Sqrt 实体，进而引发更深层次的实例化，造成实例化实体数目的爆炸（甚至实例化无法中止）。
  但是：为一个类模板定义一个 typedef 或者计算它的 sizeof 并不会导致 C++编译器实例化该实例的实体。于是就出现了一个解决方案：
  typedef typename IfThenElse<(N<mid*mid),
  Sqrt<N, LO, mid-1>,

```
                                        Sqrt<N, mid, HI> >::ResultT       SubT;
enum {result = SubT::result; };
```

这样就只会引发真正需要的实体的实例化，大幅度的减少了实例化出来的实体的数目。

- 一个 template metaprogram 可以包含以下几部分：
  - 状态变量：也就是模板参数；
  - 迭代构造：通过递归实例化；
  - 路径选择：通过使用条件表达式 IfThenElse 模板或特化机制；
  - 整型算法：一般使用枚举或静态常量；静态成员函数也是一个非常好用的保存状态的技术。
- *在实际开发中，template metaprogram 使用的很少，然后在某些情况下，template metaprogram 又是实现高效率的、对性能要求很严格的模板的一个不可替代的工具。*不过这样的高运行时效率会出的代价就是漫长的编译时间和编译期内存量。
- 递归实例化和递归模板实参会很快的将模板实例化变得难以理解，在组织递归实例化的时候，我们仍然（趋向于）避免在模板实参中使用递归嵌套的实例化。
- template metaprogram 的一个非常实用的技术：用于展开数值计算的循环（表达式模板，后面将会介绍）。
- 一个自己手写的用来输出素数的例子，很简单的表明了一个模板元编程：

```
#include <iostream>
#include <typeinfo>
#include <ctime>
using namespace std;

//DrivedTimes<N,M>表示 N 可以被<=M 的数整除的次数
template<int N, int M = N>
class DrivedTimes {
public:
  enum { drive_time = ((N % M == 0) + DrivedTimes<N, M-1>::drive_time) };
};
//DrivedTimes<N,1>的特化作为递归的结束条件
template<int N>
class DrivedTimes<N, 1> {
public:
  enum { drive_time = 1 };
};

//ShowPrime<N>表示打印所有<=N 的，并且是素数的数
template <int N>
class ShowPrime {
public:
  static void IsPrime(){
    if (DrivedTimes<N>::drive_time == 2) {
      cout << setw(5) << N << " is prime!" << endl;
    }
```

```
    ShowPrime<N-1>::IsPrime();
  }
};
//ShowPrime<2>的特化作为递归的结束条件
template<>
class ShowPrime<2> {
public:
  static void IsPrime() {
      cout << setw(5) << 2 << " is prime!" << endl;
  }
};

int DrivedTimesInNormal(int n){
  int times = 0;
  for (int i = 1; i <= n; ++i) {
    if (n % i == 0) {
      ++times;
    }
  }
  return times;
}

void ShowPrimeInNormal(int n){
  for (int i = 0; i < n; ++i) {
    if (DrivedTimesInNormal(i) == 2) {
      cout << setw(5) << i << " is prime#" << endl;
    }
  }
}

int main(){
  time_t b_t = clock();
  for (int i = 0; i < 10000; ++i) {
    ShowPrime<200>::IsPrime();
  }
  time_t e_t = clock();
  cout << e_t - b_t << endl;

  b_t = clock();
  for (int i = 0; i < 10000; ++i) {
    ShowPrimeInNormal(200);
  }
  e_t = clock();
  cout << e_t - b_t << endl;
  return 0;
}
```
当循环 10000 次时，模板元编程的版本依然几乎不需要时间，而普通的版本就需要大量的运行期时间（大约 1 秒）用来计算了，模板元编程技术带来的效率提升显而易见。

第十八章：表达式模板

- 表达式模板技术刚开始是为了支持一种数值数组的类 Array 而引入的。类似的数值数组中，如果既希望使用紧凑的运算符写法，又希望得到很高的效率，就不是容易的事情，这时候就需要运用表达式模板技术来进行编译期的扩展。
- 得到高效率的最好办法是手工编写循环，最大限度的减少临时对象的生成。

  ```
  for(int i = 0; i < x.size(); ++i) {
    x[i] = 1.2*x[i] + x[i]*y[i];
  }
  ```

  然而手工编写循环使得语法并不是那么优雅，一般的库使用者希望使用简单的运算符语法：

  ```
  x = 1.2*x + x*y;    //但如果未经特殊优化，这样会产生大量的临时对象
  ```

  表达式模板技术就是把右边的表达式转换为如下的模板类型：

  ```
  A_Add< A_Mult< A_Scalar<double>, Array<double> >,
              A_Mult< Array<double>, Array<double> > >
  ```

  再通过在 Array 的赋值运算符中通过循环去展开模板的取下标操作，如果编译器能够优化掉这过程中产生的这些小对象和内联的话（理论上编译器不难做到），这样就能产生出与手工代码循环效率相媲美的代码。
- 下面对表达式模板的工作过程作一个简单的分析：x = 1.2*x + x*y;
  - 首先 1.2 会被隐式转换为 A_Scalar<double>
  - 然后和 x 一起应用于 operator*运算符，返回一个 Array<double, A_Mult<double, A_Scalar<double>, SArray<double> > >的模板类
  - 同样 x*y 也会被应用于 operator*运算符，生成一个 Array<double, A_Mult<double, SArray<double>, SArray<double> > >的模板类
  - 这两个生成的模板类被应用于 operator+运算符，最后生成 Array<double, A_Add<double, A_Mult<double, A_Scalar<double>, SArray<double> >, A_Mult<double, SArray<double>, SArray<double>>>>的类型
  - 需要注意的一点是：这里的 Array、A_Multi、A_Add 类型的取下标算符都不会引发数据的拷贝和临时对象的产生，这里都是产生一个到原始数据的引用。
  - 最后的赋值运算符 operator=会应用于=号左边的 x（类型 Array<double, SArray<double>>和右边的最后被生成的类型，**在 Array 模板类的成员模板函数 operator=中，进行循环并展开表达式**：

    ```
    template<typename T2, typename Rep2>
    Array& operator= (Array<T2, Rep2> const& b) {
    ```

```
        for (size_t idx = 0; idx<b.size(); ++idx) {
            expr_rep[idx] = b[idx];
        }
        return *this;
    }
```

- 当调用到`expr_rep[idx] = b[idx];`时，就会展开右边的运算符[]。这时，通过 b[idx]的展开，右边的类型 Array<double, A_Add<double, A_Mult<double, A_Scalar<double>, SArray<double> >, A_Mult<double, SArray<double>, SArray<double>>>>就会被层层展开。由于右边的类型里都是内联函数，且都是返回原始数据的引用，这样，`expr_rep[idx] = b[idx];`就会被展开成类似于手工循环的表达式。
- *因此，如果在编译器对内联函数和小对象优化做的比较好的情况下，表达式模板技术能够取得和手工写循环近乎相同的效率（同时又获得了紧凑的语法）。*

• C++标准库中包含了一个名为 valarray 的类模板，它主要是用于实现我们在本章中开发 Array 模板时所用到的一些技术。不过 valarray 是一个失败品，它并没有完成计划的功能，对于标准库中的 valarray，如果执行原先所设计的操作，效率是相当低的。

所以《C++标准程序库》一书中也建议不要使用 valarray，也许标准库以后会出现更好的使用了表达式模板技术的新数值数组类模板吧。

第四部分：高级应用程序

很多时候需要对程序本身进行编程，但是 C 预处理器的功能很有限，远远不能胜任我们日常编程的要求。对于一些相对较小、并且互相独立的简单功能，模板是最好的实现方式，这部分将会讨论一些简单的、常见的模板的用途：

- 一个用于区分类型的框架
- 智能指针
- tuple
- 仿函数

第十九章：类型区分

- 对于一个模板参数，如果能知道它究竟是内建类型、指针类型、class 类型等等是非常有用的。可以想象，把它是否是某一种类型当参数传给函数或类型，可以针对这种特殊情况做特殊的处理，并从中获得好处。一个简单的例子就是 copy 模板函数，如果能通过对类型进行分析，发现它可以有 bitwise 语意，在进行 copy 的时候，就可以直接使用 mem_copy 得到最高的效率，而不是去调用它的 operator=，这肯定比位拷贝慢太多了。
- 分辨出基本类型是很简单的，因为基本类型的数目是有限的，对它们一一进行特化就可以简单直接的识别出来。*虽然这样一一特化的方式可以无比正确的分辨出基本类型，但是手段略显笨拙，是否有其它更漂亮的方法呢？*
- 组合类型是指一些构造自其它类型的类型。简单的组合类型包括：普通类型、指针类型、引用类型、数组类型。
  识别组合类开可以使用相应的特化来解决，如：
  ```
  template<typename T>
  class CompoundT<T&> {...}  //识别引用类型，指针类型、数组类型、成员指针类似
  ```
- 分辨函数类型有 2 种常见的手段：
  - 使用针对函数的特化语法：
    ```
    template<typename R>
    class Compoud<R()> {...}  //类似的一系列特化
    template<typename R, typename P1>
    class Compoud<R(P1, ...)> {...}  // 类似的一系列特化
    ```
  - 利用 SFINAE 原则：因为数组的元素不是为 void 值、引用或者函数（void,引用在前面已经可以被正确的识别了）。

```
template<typename U> static char test(...);
template<typename U> static int test(U (*)[1]);
enum { Yes = sizeof(test<T>(0) == 1 };
```

  - 还有一些其它的、不常用的手法也可以解决。

- SFINAE 原则的使用时机：如果我们需要识别 X，如果能找到一种构造对于 X 是无效的，但是对其它类型都是有效的；或者完全相反。这样就可以利用 SFINAE 原则进行正确的识别。SFINAE 原则对于模板编程来说，真是非常的重要的！
- 分辨出枚举类型：利用从枚举类型到整型的隐式转型，就能够识别出枚举类型。具体分辨方法为：如果 T 是非基本类型、非指针、非引用、非成员函数指针时（前面已经识别），还能够被隐式的转换为整型，那么 T 就一定是枚举类型。

```
enum {
  Yes = IsFundaT<T>::No &&
      !CompoundT<T>::IsRefT &&
      !CompoundT<T>::IsPtrT &&
      !CompoundT<T>::IsPtrMemT &&
      sizeof(enum_check(ConsumeUDC<T>())) == 1
};
```

而且，即使 class 类型定义了一个到 int 类型的转换也不要紧，因为 C++规定了对于以匹配为目的的隐式转型的话，最多只能自动转型一次。于是，就巧妙的避开了 class 的可能的转换，于是这样的检测就一定能检测出枚举类型。

- 分辨 class 类型：直接使用排除法就可以了，如果一个类型不是基本类型、不是枚举类型、不是组合类型，那就一定是 class 类型（而前面的讨论已经可以分辨出那些非 class 类型了）。
- 对于某个实体，这种能够在程序中获它的高层次属性（诸如类型结构）的能力通常称为反射。这一章所讨论的反射属于"编译期反射"，像 C#、Java 的反射就是"运行期反射"。与编译期多态和运行期多态一样，编译期反射没有运行期的代价，执行速度会很快，但是编译时间又变长了。
- boost 的 Types Traits 库已经实现了一个相当完整的类型区分模板，功能比这里讨论的强大太多了，需要的时候可以直接拿来用（C++11 中也已经有了）。
- 最后把这一章所讨论的代码汇集一下：

```
// primary template: in general T is no fundamental type
template<typename T>
class IsFundaT {
public:
  enum {
    Yes = 0, No = 1
  };
};

// macro to specialize for fundamental types
#define MK_FUNDA_TYPE(T) \
```

```
template<> \
class IsFundaT<T> \
{ \
public: \
  enum { Yes = 1, No = 0 }; \
};
MK_FUNDA_TYPE(void)
MK_FUNDA_TYPE(bool)
MK_FUNDA_TYPE(char)
MK_FUNDA_TYPE(signed char)
MK_FUNDA_TYPE(unsigned char)
MK_FUNDA_TYPE(wchar_t)
MK_FUNDA_TYPE(signed short)
MK_FUNDA_TYPE(unsigned short)
MK_FUNDA_TYPE(signed int)
MK_FUNDA_TYPE(unsigned int)
MK_FUNDA_TYPE(signed long)
MK_FUNDA_TYPE(unsigned long)
#if LONGLONG_EXISTS
MK_FUNDA_TYPE(signed long long)
MK_FUNDA_TYPE(unsigned long long)
#endif  // LONGLONG_EXISTS
MK_FUNDA_TYPE(float)
MK_FUNDA_TYPE(double)
MK_FUNDA_TYPE(long double)
#undef MK_FUNDA_TYPE

template<typename T>
class CompoundT {           // primary template
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
    IsFuncT = 0, IsPtrMemT = 0
  };
  typedef T BaseT;
  typedef T BottomT;
  typedef CompoundT<void> ClassT;
};

template<typename T>
class CompoundT<T&> {       // partial specialization for references
public:
  enum {
    IsPtrT = 0, IsRefT = 1, IsArrayT = 0,
    IsFuncT = 0, IsPtrMemT = 0
  };
  typedef T BaseT;
  typedef typename CompoundT<T>::BottomT BottomT;
  typedef CompoundT<void> ClassT;
};
```

```
template<typename T>
class CompoundT<T*> {       // partial specialization for pointers
public:
  enum {
    IsPtrT = 1, IsRefT = 0, IsArrayT = 0,
    IsFuncT = 0, IsPtrMemT = 0
  };
  typedef T BaseT;
  typedef typename CompoundT<T>::BottomT BottomT;
  typedef CompoundT<void> ClassT;
};

template<typename T, size_t N>
class CompoundT<T[N]> {    // partial specialization for arrays
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 1,
    IsFuncT = 0, IsPtrMemT = 0
  };
  typedef T BaseT;
  typedef typename CompoundT<T>::BottomT BottomT;
  typedef CompoundT<void> ClassT;
};

template<typename T>
class CompoundT<T[]> {    // partial specialization for empty arrays
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 1,
    IsFuncT = 0, IsPtrMemT = 0
  };
  typedef T BaseT;
  typedef typename CompoundT<T>::BottomT BottomT;
  typedef CompoundT<void> ClassT;
};

template<typename T, typename C>
class CompoundT<T C::*> {  // partial specialization for pointer-to-members
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
    IsFuncT = 0, IsPtrMemT = 1
  };
  typedef T BaseT;
  typedef typename CompoundT<T>::BottomT BottomT;
  typedef C ClassT;
};

template<typename R>
```

```
class CompoundT<R()> {
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
    IsFuncT = 1, IsPtrMemT = 0
  };
  typedef R BaseT();
  typedef R BottomT();
  typedef CompoundT<void> ClassT;
};

template<typename R, typename P1>
class CompoundT<R(P1)> {
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
    IsFuncT = 1, IsPtrMemT = 0
  };
  typedef R BaseT(P1);
  typedef R BottomT(P1);
  typedef CompoundT<void> ClassT;
};

template<typename R, typename P1>
class CompoundT<R(P1, ...)> {
public:
  enum {
    IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
    IsFuncT = 1, IsPtrMemT = 0
  };
  typedef R BaseT(P1);
  typedef R BottomT(P1);
  typedef CompoundT<void> ClassT;
};

struct SizeOverOne {
  char c[2];
};

template<typename T,
    bool convert_possible = !CompoundT<T>::IsFuncT &&
        !CompoundT<T>::IsArrayT>
class ConsumeUDC {
public:
  operator T() const;
};

// conversion to function types is not possible
template<typename T>
class ConsumeUDC<T, false> {
```

```
};

// conversion to void type is not possible
template<bool convert_possible>
class ConsumeUDC<void, convert_possible> {
};

char enum_check(bool);
char enum_check(char);
char enum_check(signed char);
char enum_check(unsigned char);
char enum_check(wchar_t);

char enum_check(signed short);
char enum_check(unsigned short);
char enum_check(signed int);
char enum_check(unsigned int);
char enum_check(signed long);
char enum_check(unsigned long);
#if LONGLONG_EXISTS
char enum_check(signed long long);
char enum_check(unsigned long long);
#endif  // LONGLONG_EXISTS
// avoid accidental conversions from float to int
char enum_check(float);
char enum_check(double);
char enum_check(long double);

SizeOverOne enum_check(...);
// catch all
template<typename T>
class IsEnumT {
public:
  enum {
    Yes = IsFundaT<T>::No &&
        !CompoundT<T>::IsRefT &&
        !CompoundT<T>::IsPtrT &&
        !CompoundT<T>::IsPtrMemT &&
        sizeof(enum_check(ConsumeUDC<T>())) == 1
  };
  enum {
    No = !Yes
  };
};

template<typename T>
class IsClassT {
public:
  enum {
    Yes = IsFundaT<T>::No &&
```

```
        IsEnumT<T>::No &&
        !CompoundT<T>::IsPtrT &&
        !CompoundT<T>::IsRefT &&
        !CompoundT<T>::IsArrayT &&
        !CompoundT<T>::IsPtrMemT &&
        !CompoundT<T>::IsFuncT
  };
  enum {
    No = !Yes
  };
};

// define template that handles all in one style
template<typename T>
class TypeT {
public:
  enum {
    IsFundaT = IsFundaT<T>::Yes,
    IsPtrT = CompoundT<T>::IsPtrT,
    IsRefT = CompoundT<T>::IsRefT,
    IsArrayT = CompoundT<T>::IsArrayT,
    IsFuncT = CompoundT<T>::IsFuncT,
    IsPtrMemT = CompoundT<T>::IsPtrMemT,
    IsEnumT = IsEnumT<T>::Yes,
    IsClassT = IsClassT<T>::Yes
  };
};

template<typename T>
void check() {
  if (TypeT<T>::IsFundaT) {
    std::cout << " IsFundaT ";
  }
  if (TypeT<T>::IsPtrT) {
    std::cout << " IsPtrT ";
  }
  if (TypeT<T>::IsRefT) {
    std::cout << " IsRefT ";
  }
  if (TypeT<T>::IsArrayT) {
    std::cout << " IsArrayT ";
  }
  if (TypeT<T>::IsFuncT) {
    std::cout << " IsFuncT ";
  }
  if (TypeT<T>::IsPtrMemT) {
    std::cout << " IsPtrMemT ";
  }
  if (TypeT<T>::IsEnumT) {
    std::cout << " IsEnumT ";
```

```
  }
  if (TypeT<T>::IsClassT) {
    std::cout << " IsClassT ";
  }
  std::cout << std::endl;
}

// check by passing type as function call argument
template<typename T>
void checkT(T&) {  //注意这里应该用 T&，书上错误的使用了 T。因为引用类型传递时才不会引发实参的 decay，否则如果值传递的话引起实参的 decay。就会错误的把函数 decay 成函数指针；把数组类型 decay 成指针
  check<T>();

  // for pointer types check type of what they refer to
  if (TypeT<T>::IsPtrT || TypeT<T>::IsPtrMemT) {
    check<typename CompoundT<T>::BaseT>();
  }
}

class MyClass {
public:
  enum YesOrNo {
    Yes, No
  };
  void f();
  int mem;
};

int main(int argc, char** argv) {
  cout << boolalpha << bool(TypeT<int>::IsFundaT) << endl;
  cout << boolalpha << bool(TypeT<int&>::IsRefT) << endl;
  cout << boolalpha << bool(TypeT<int*>::IsPtrT) << endl;
  cout << boolalpha << bool(TypeT<MyClass>::IsClassT) << endl;
  cout << boolalpha << bool(TypeT<MyClass::YesOrNo>::IsEnumT) << endl;
  cout << boolalpha << bool(TypeT<int MyClass::*>::IsClassT) << endl;
  cout << boolalpha << bool(TypeT<void (MyClass::*)()>::IsPtrMemT) << endl;
  return 0;
}
```

输出显示这种编译期反射机制已经在正常工作了！

第二十章：智能指针

- 就 C++而言，智能指针是一些在行为上类似于普通指针的类（因为提供了取引用运算符->和*），而且该类还封装了一些内存管理或资源管理 policy。

- 有两种不同的内存所有权模型－－独占与共享：
  - 独占(scoped_ptr)：几乎没有开销，可以处理好异常抛出；
  - 共享(shared_ptr)：会导致非常复杂的对象生命周期，不过建议让程序自身来处理对象的生命期；
- holder 和 trule：holder 类型独占一个对象；而 trule 可以使对象的拥有者从一个 holder 传递给另一个 holder。
- 析构函数绝对不能抛出异常：当一个异常被抛出的时候，析构函数都是被自动调用的，而此时如果再抛出另外一个异常，那么将会导致程序立即终止。
- **显式转型和隐式转型之间的细微区别**：
  - 显式转型，X y(x); ，显式调用拷贝构造函数，因为可以接受 explicit 的拷贝构造函数；
  - 隐式转型，X y=x; ，隐式调用拷贝构造函数，要求拷贝构造函数不能有 explicit 修饰。
- 资源获取即初始化 RAII，这种手法已经被无数的经典书籍讨论过了。

  Holder<MyClass> p(new MyClass());
- 从智能指针获取原始指针的两种方式：
  - &*smart_point;
  - smart_point.operator->();
  - 如果使用了后一种方法，那么就意味着已经做了一些比较危险的操作。
- 很明显，Holder 由于一定会在析构函数中 delete 对象，所以它只能独占对象，那么它的拷贝和复制操作就显然是不可行的，否则会扰乱 Holder 所拥有的对象的 delete 机制。

  所以一个 holder 在内存中只能存在一份，进而如果一定需要在参数中传递 holder，那么也只能以引用的形式传递。
- trule(transfer capsule)就是一个专门用来传递 holder 的辅助类模板。

  trule 的原理在于包括 2 个隐式转型：从 holder => trule => holder。在这个 2 层的转换过程之中，trule 会自动处理好期间指针的转移过程（先从开始的 holder 把资源指针临时存放在 trule 中，再交给最后的 holder，完全整个转移过程）。

```
class trule<T> {
public:
    trule(holder<T> const &){...}
}
class holder<T>{
public:
    holder(trule<T> const &){...}
}
```

这样，就可以使用 trule 来作为需要返回 holder 的函数的返回类型了，就可以通过 trule 实现 holder 的跨函数传递。

- 对于动态分配对象的管理，一条普遍的规则可以简单阐述如下：*对于任何一个动态分配的对象，如果没有任何变量指向它，那么这个对象就应该被销毁，其占有的内存同时也应该被释放。*
由这种想法衍生出来的一种思想就是引用计数：对于每个动态分配的对象，保存一个计数用于代表指向该对象的指针的个数，当计数值减少到 0 时就删除此对象。
- 实际上，那个计数器的内存分配都会使用专用的内存分配器，这种分配器专门用于分配大小固定的小对象。
- 计数器的并发访问：在多线程的环境中，计数器也可能会被多个线程同时访问，所以还要为计数器本身增加一些并发控制机制，实际中都需要某种形式的锁来实现这个功能。
- 鉴于使用非标准方式释放对象的情况是肯定存在的，所以引入一种单独的对象释放的 policy 是很有必要的。很明显 OpenCV 中就存在大量的非标准方式释放内存，所以正常的智能指针都会有释放内存的 policy。
- 智能指针实现中需要注意的几点：
    - 在拷贝赋值操作中，要判断是否为自赋值；
    - 由于空指针并没有一个可关联的计数器，所以在减少计数值之前，必须先显式地检测空指针的情况。
- 计数器的类型，非侵入式和侵入式：
    - 非侵入式计数：把计数器分配在对象之外，一般都是采用这种非侵入式；
    - 侵入式计数：计数器就在对象的内部，所以需要在类一开始设计的时候就考虑了计数器，所以一般用于专用的对象。但是，这种在内部的计数器实现起来更简单、更快捷。
- 常数性：Ptr<T>的构造函数应该接受 T*类型，而不是 T const *类型，这是一个常犯的错误！T*类型不能由 T const*类型来赋值，对于希望有常数性的使用，应该是 Ptr<T const>，这样在构造函数里就被自动的扩展为 T const *了。
- 智能指针的隐式转型是一个非常深奥的问题：因为普通指针支持到 void*、基类指针、bool 类型的隐式转型，所以我们希望智能指针也能模拟这个特性。
    - 首先最容易想到的是提供一个到原始指针的隐式转型，但这是不可行的。因为首先这样就破坏了"所有指针这个对象的指针都有引用计数"这个最基本的假设。并且使得很多指针操作可以被施加于智能指针之上了，如 delete p; p[4];等等。
    隐式转型往往是可怕的，会带来很多难以意料的问题。
    - 解决到 void*和基类指针的转换问题可以通过提供一个使用不同类型的隐式拷贝构造函数来解决，并在这个隐式拷贝构造函数中转换原始指针的类型。如果原始指针的类型不匹配的话，那么模板的实例化就会产生错误，保证了安全性。
    与转型运算符相比，转型构造函数更容易实现这种所希望的隐式转型。

- 对于到bool类型的隐式转型，简单的方案是定义到bool或void*的隐式转换。但就像前面所说的，隐式转型往往是可怕的，几乎一定会带来不可意料的效应。
  更好的解决方案是定义一个到成员指针类型的隐式转型。**成员指针也可以被隐式转到bool，但是它与普通指针的区别在于：它不能被delete和运用指针运算。**

  ```
  template<typename T>
  class Ptr{
   class BoolConversionSupport{
    int dummy;
   }
  public:
   operator BoolConversionSupport::*() const {
    return this->object_point_to ? &BoolConversionSupport::dummy : 0;
   }
  }
  ```

  而且由于只需要声明这个内部的类类型，而不需要它的任何实例，所以这样做其实是没有代价的。

- 原始指针还支持相等比较（=, !=）和排序比较（<, <=, >, >=），但是一般来说智能指针只应该实现相等比较，因为智能指针总是指向单个对象或者数组的开头，并不关心顺序。

  ```
  template<typename T>
  class Ptr{
  public:
   friend bool operator=(Ptr<T> const &cp, T const *p){
    return cp == p;
   }
  }
  template<typename T1, typename T2>
  bool operator=(Ptr<T1> const &cp1, Ptr<T2> const &cp2){
   return cp1.operator->() == cp2.operator->();
  }
  ```

  一般来说直接使用oprator->()是不安全的， 但是这里把它的使用范围封闭在函数的内部是可行的，这样就直接使用内部的指针实现了比较操作（如果需要排序比较也类似）。

- C++标准库中的auto_ptr也是独占模式，类似于holder/trule的手法，但是由于在变量初始化上下文中，auto_ptr使用了C++的一些重载规则，所以auto_ptr并不需要第2个模板trule，然而这种做法充满争议（很明显它并不好用、也充满着问题，所以C++11删除了它）。

第二十一章：tuple

- 简单的 duo 类就是与 std::pair 非常类似的类模板，它只能容纳 2 个不同类型的元素。
- 可递归的 duo： 简单的给 duo 添加一个递归的模板就可以实现递归的定义

```
template <typename T1, typename T2, typename T3>
class duo<T1, duo<T2, T3>> {...}
```

不过也需要注意递归的结束条件

```
template <typename T>
class duo<T, void> {...}
```

- 然后可以写一个类似于 get<3>(duo), getT<3>(duo)的函数来获取一个可递归的 duo 的第 N 个元素的值和它的类型。getT 采用的手法类似如下，可以想象 std::get<3>(tuple)也应该是类似实现的：

```
template <int N, typename T>
class duoT {
public:
      typedef void ResultT;    //默认情况下都返回 void 类型
};
template <typename A, typename B>
class duoT<1, duo<A,B>> {       //针对最普通的，只有 2 个元素的特化
public:
      typedef A ResultT;
};
template <typename A, typename B>
class duoT<2, duo<A,B>> {       //针对最普通的，只有 2 个元素的特化
public:
      typedef B ResultT;
};
template <int N, typename A, typename B, typename C>
class duoT<N, duo<A, duo<B, C>>> {             //递归的定义
public:
      typedef typename duoT<N-1, duo<B, C>>::ResultT ResultT;
};
template <typename A, typename B, typename C>
class duoT<1, duo<A, duo<B, C>>> {             //递归的结束条件
public:
      typedef A ResultT;
```

```
};
template <typename A, typename B, typename C>
class duoT<2, duo<A, duo<B, C>>> {           //递归的结束条件
public:
      typedef B ResultT;    //可以直接返回 B 的原因是：duo 都是第 2 个参数递归的
};
```

- NullT 技巧：因为 void 不能用来创建对象，即不存在 void t;所以经常使用一个空类 NullT 来代表 void。

  class NullT{};

- 包装成 Tuple：duo 的使用非常不方便，如果需要创建一个 3 个元素的 duo，就需要这样写：

  duo<int, duo<char, double>> d;

  所以，需要用一些技巧来实现方便的使用 duo，当然，方便就会付出一些代价。

  书上使用了 Tuple 类模板来使用 duo，但是 tuple 的元素个数是有限制的：

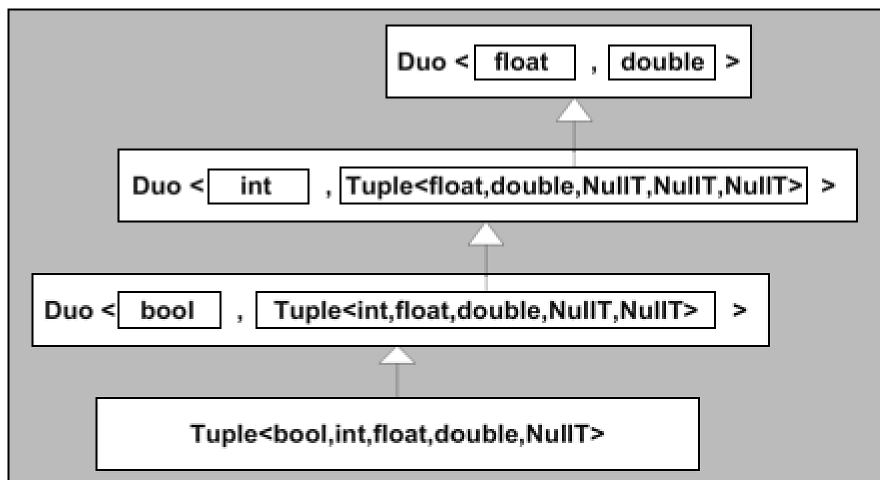
```
template<typename P1,
         typename P2 = NullT,
         typename P3 = NullT,
         typename P4 = NullT,
         typename P5 = NullT>
class Tuple : public Duo<P1, typename Tuple<P2,P3,P4,P5,NullT>::BaseT>
{...};
```

为什么 Tuple 要继承自 Duo，而不是支持在内部使用 5 个 Pn，原因还在在于编译器的空基类优化。如果使用普通的内部 5 个 Pn 成员，那么每个成员都需要付出代价，即使这个成员是 NullT 类型也一样；但是如果使用继承的手法，就可以利用空基本优化掉任何 NullT 类（这本来就不是该付出的代价）。

同时还要注意它的递归结束条件：

```
template<typename P1, typename P2>
class Tuple<P1, P2, NullT, NullT, NullT> : public Duo<P1, P2>{...};
```

最后的类图结构会是这样的(对于 Tuple<bool,int,float,double> t4)：



然后再添加 make_tuple 包装函数方便使用。这样，最后得到的 Tuple 类就和 C++11 中的 std::tuple 非常类似了，有了这样的 tuple，一些简单的结构包装是很方便的。

- Andrei Alexandrescu 在《Modern C++ Design》中开发了一种有趣的实现，他把 tuple 的类型列表（list of type）从 tuple 的域列表（list of field）中分离出来，从而就有了 type list 的概念，而且在该书中 type list 有着多种应用（其中一个应用就是实现一个封装类型的 tuple 构造）。

第二十二章：函数对象和回调

- 函数对象：可以使用函数调用语法进行调用的任何对象。

  C 语言中可以使用 3 种实体进行调用，函数、宏、函数指针，但只有函数指针是对象。

  C++中多了，重载了函数调用操作符的 class、函数引用（很少见）、成员指针等等。
- 对于仿函数而言，几乎所有的使用形式都是某种形式的回调：对于一个程序库，它的客户能够使库调用客户端自定义的某些函数，这种方式就叫作回调。
- 函数对象和仿函数的确切概念区别：函数对象就是重载了函数调用操作符的 class（*函数对象一定是一个 class*）；而仿函数包含了所有能够以函数语法进行调用的对象。

  所以：仿函数包含函数对象。
- 直接调用、间接调用、内联调用：
  - 直接调用是指在**编译期可以确定目标的调用**，就可以直接直接调用（例如普通函数调用）；
  - 间接调用是指在**编译期无法确定确切的目标的调用**，只能在运行期进行目标确定的调用（例如使用函数指针进行调用）；
  - 内联调用是把**函数的代码在调用点直接展开**。
  - 效率很明显为：内联调用>直接调用>间接调用，它们之间的效率差距是很大的。
  - 随着计算机体系结构的不断复杂化，直接调用和间接调用之间的效率差距也不断增大。因此，

编译器通常都会尽可能地生成直接调用的指令。

- 通过使用内联，就可以大大的帮忙编译器进行优化。所以，应该充分认识到内联函数的好处。通过模板来使用回调的话 ，可以大大的加快调用的速度。因为模板是编译期的东西，所以使用模板来生成机器码的话，那这些机器码将主要是直接调用和内联调用，所以可以被编译器大大的加速。而如果使用传统的回调的话，那么将产生间接调用（函数指针），这可比普通的函数调用慢的多。
- 函数指针和函数引用：（我们平常使用的都是函数指针，但函数引用也是存在的）
  - 函数指针：void (*pf)();　　　　　pf = foo;　//隐式 decay
  - 函数引用：typedef void FooT();　FooT &rf=foo;
  - 函数名 foo 实际上是一个左值，因为它可以被绑定到 non-const 引用(rf)，但是这个左值却是不可以修改的。
- typeid 运算符和 sizeof 运算符都不会去它的参数进行 decay，都会返回最原始的类型。**实际上，一般 typeid 和 sizeof 都不会去使用实体**，只有在 typeid(d)中的 d 是一个多态类型对象时（因为通常而言，我们要等到运行期才能确定多态 typeid 运算符的结果），typeid 才会去使用 d。
- 成员函数指针与普通的指针是有本质区别的，成员函数指针更多的是表示一种位移。现今的许多编译器都对成员函数指针使用了 3 - 值结构，分别为：
  - 成员函数的地址，如果是一个虚函数，那么该值为 NULL
  - 基于 this 的地址调整
  - 一个虚函数索引
- class 类型的仿函数：如果使用重载了函数调用运算符的 class 类型对象的话，可以给我们带来很多好处：譬如灵活性、性能，甚至二者兼备。
  - 因为 class 类型的仿函数可以很好的利用 inline 函数提供的优化，使得 C++的 std::sort 的速度经常会比 C 语言的 sort()还要快，这其实是编译器替我们展开了函数调用。
  - 如果一个 class 类型的仿函数并没有包含任何状态的话，那么它的行为完全由它的类型所决定的。于是，我们可以以模板实参的形式来传递该类型，用于自定义程序库的组件行为。
- 指定仿函数的形式：可以使用模板类型的实参、函数调用实参等等方便指定。
  - 模板类型的实参：std::sort<std::less<...>>(...);
    这种形式的调用中对 std::less 的选择是模板参数，它会在编译期进行展开。并且由于 std::less 是内联函数，所以一个优化的编译器能够产生本质上等价于不使用仿函数，直接手工编译代码的效率。这是非常诱人的！但是，它只能传递函数对象。
  - 函数调用的实参：std::sort(..., std::less<...>());
    C++库中的 std::sort 就是这样传递的，这种技术的优点还在于可以传递指针进行调用，所有的仿函数都可以被传递进去。这种形式的效率和上一种相同，同时还支持传递函数指针，

所以在标准库中得到了广泛应用。

- ○ 还可以作为非类型模板实参：*并不常用，class 类型的仿函数并不适合以非类型模板实参的形式进行传递。*

  而且 C++中借助引用或者指针的非类型实参必须能够和参数类型精确匹配，从派生类到基类的转型是不允许的，而进行显式类型转换也会使实参无效，同样也是错误。

  class Base; class Derived : public Base;

  Base b; Derived d;

  template<Base &> void sort(...);

  sort<b>(...);

  sort<d>(...); //错误，必须用 sort<(Base&)d>(...);

- ○ 不能把一个具有 C 链接的实体传递给模板的非类型模板参数。

- ○ 模板是一种编译期机制，基于这种原因，对于初看起来是一个间接调用的函数（因为涉及到指针），大多数编译器实现都能把这种间接调用转化为直接调用。而且，如果所调用的函数是内联函数，而且它的定义在调用点是可见的，那么期望这种调用是内联调用同样也是合理的。

- 在程序设计的上下文中，内省指的是一种能够查看自身的能力（类似反射？）。
- 纯仿函数：没有任何副作用的仿函数，我们通常把它称为纯仿函数。

  例如，通常而言排序规则就必须是纯仿函数，否则的话排序操作的结果将会是毫无意义。
- 值绑定：有点类似 std::bind1st, std::bind2nd，但是实现的比标准库中的 bind 要强大一些。

  标准库中的 std::bind1st, std::bind2nd 要求被绑定的函数对象必须继承自 std::binary_function<argT1, argT2, returnT>，这样就可以直接获得了所有的函数签名参数，实现绑定就容易太多了。而且因为继承自 bindnary_function 所以只允许有 2 个参数。

  我自己简单的实现了一个类似于标准库的值绑定如下：

```
class Add : public binary_function<int, int, void>{
public:
	void operator()(int a, int b) const {
		cout << a + b << endl;
	}
};
template<typename T>
class mybinder1st {
public:
	mybinder1st(T const &op, typename T::first_argument_type const
&arg1)
		: op_(op), arg1_(arg1)
	{}
```

```
	typename T::result_type operator()(typename T::second_argument_type
const
&arg2){
		return op_(arg1_, arg2);
	}
private:
	T const &op_;
	typename T::first_argument_type const &arg1_;
};
template<typename TOp, typename TArg>
mybinder1st<TOp> mybind1st(TOp const &op, TArg const &arg) {
	typedef typename TOp::first_argument_type _Arg1_type;
	return mybinder1st<TOp>(op, _Arg1_type(arg));
}
int main() {
	mybind1st(Add(), 2)(5);
	return 0;
}
```

正确的输出：7

书上实现的 bind 功能更强大一些，也要求所有的函数对象都提供自身的参数信息（要求提供了 ReturnT 和 Param*T，貌似 boost::bind 似乎什么都不要求，太强了），不过提供了对普通函数指针的包装 func_ptr，使得更通用一些。

boost::bind 实现的比标准库和书上的绑定机制都强大太多了，它太好用了！一定要找时候研究一下 boost::bind 的实现机制和源代码。

- 谓词、断言（predicates）：返回 bool 值的函数或函数对象，通常而言 predicates 必须是纯仿函数，否则将出现不可预料的结果(The C++ Standard Library 和 Effective STL 中也反复强调过)。
- 很明显 C++标准库中提取的绑定机制并不完善，boost 库提供了很好的扩展适配器 boost::bind，boost 值得我们去学习的。

附录 A：一处定义原则

- C++程序设计体系中的一块基石就是**一处定义原则（ODR）**：
  对于同一程序，非内联函数只能在所有的头文件中定义一次；对于类和内联函数，每个翻译单元最多只能定义一次，并且就确保相同实体的所有定义都是相同的。
- 翻译单元：一个翻译单元直观的说就是一个.cpp 文件，包括所有它所引用的.h 文件。
- 声明和定义：在 ODR 的上下文中，分清声明和定义的概念非常重要。
  声明是一种"把一个 C++名称引入或者重要引入到程序"的构造。一个声明也可以是一个定义，这取决于它所引入的是哪些实体以及如何引入这些实体的。（*前面讨论过声明和定义*）
- 内部链接的实体：就是在匿名的 namespace 中定义或者加了 static 修饰符的实体。
- 和 class 类型不同的是，内联函数的定义可以位于使用点之后，例如：

```
inline void foo();
int main(){
  foo();
}
inline void foo() {...}
```

  不过，这种代码尽管是有效代码，但是一些编译器可能不能内联这样在调用时看不到定义的函数调用，从而不能获得预期的效果。
- 跨翻译单元的等价性约束：能够在多个翻译单元中定义的实体（类、内联函数）可能会不匹配，但是传统的编译技术很难检测（因为编译工作是分翻译单元的）。所以 C++标准也没有强求编译器，但是要求给出"未经定义的行为"这个信息。
  当一个实体在多个位置都定义时，它们必须由完全相同的 **标记序列** 组成，一个错误的例子：

```
//翻译单元 1
static int counter = 0;
inline void increase_counter(){
  ++counter;
}
//翻译单元 2
static int counter = 0;
inline void increase_counter(){
  ++counter;
}
```

  这是错误的！因为虽然 increase_counter 都定义内部相同，但是它们却引用了不同的实体 counter，所以它们的定义实际上是不等价的。

跨翻译单元约束不仅适用于在多个位置定义的实体，也适用于声明中的缺省实参。

在模板的情况下，需要分清2个时间点：非依赖型名称在模板的定义处进行绑定时，和依赖型名称在实例化点（POI）时进行绑定，这2次绑定都必须是等价的。

附录 B：重载解析

- 重载解析规则大多数情况下，都是符合直观选择的，实际上它的目的也就是试图模拟直观选择。
- 首先，如果通过函数指针或成员函数指针来进行调用，就不会进行重载解析；因为究竟调用哪个函数是在运行期由指针（实际所指向的对象）来决定的。另外，类似函数的宏不能被重载，但不会进行重载解析了。
- 从较高的抽象层次来看，对于一个命名函数的调用，通过是通过如下的过程：
  - 查找名称，从而形成一个初始的重载集；
  - 如果有必要的话，会用各种方法对这个集合进行修改（例如发生在模板演绎的时候）；
  - 删除任何与调用不匹配（即使考虑了隐式转型和缺省实参之后仍不匹配）的候选函数，得到"可行的候选函数集"；
  - **执行重载解析来寻找最佳候选函数**，如果找不到唯一的、最佳的匹配函数，则报告二义性；
  - **检测这个被选中的最佳候选函数**。（例如如果它具有不能访问的私有成员，则出错）。
- **名称查找发生在类型检查之前**：从上面的重载过程可以看到，最后的 2 步是先寻找最佳候选函数，再检查它。

  这样就会有一个这样的问题，如果最佳匹配的候选函数不能通过最后一步的检测，但是次佳匹配的候选函数能通过最后一步的检测，那也无法匹配成功。因为次佳匹配的候选函数根本不会被检测，在最佳候选函数被检测失败之后就直接报告错误了。
- 对于 int 类型的实参，有 3 种参数类型可以与它获得完美的匹配：int, int&, int const &。

  同样可以扩展到，对于类型 T，有 3 种类型的完美匹配：T, T&, T const &
- 如果实参是一个左值，那么将会优先考虑没有 const 的版本；而对于作为右值的实参，将会优先考虑 const 版本。
- 仿函数和代理函数：

```
typedef void FuncType(double, int);
class IndirectFunction {
public:
    operator()(double, double);
    operator FuncType*() const;   //代理函数（哑函数）
}
void active(IndirectFunction const &obj) {
    obj(3.1, 5);                  //错误：二义性调用
}
```

  因为 IndirectFunction 的 2 个 operator 成员会将 2 个候选函数添加到重载集合中，其中

operator FuncType*() const; 给重载集合添加叫作代理函数（也称哑函数）。

代理函数的匹配度不如直接的 operator()，不过上面的例子中，operator()的参数与调用不完全匹配，但是代理函数的参数完全匹配，所以 2 者都需要一次转型，所以它们的优先度是相同的，因此也就产生了二义性。

- 其它的会引发重载情况：
  - 需要函数地址的时候，也会引发重载

    ```
    int (*pf)(double, double) = functions;  //引发对 functions 函数集合的重载解析
    ```
  - 另一种要求进行重载解析的情况发生在初始化的时候

    ```
    struct BigNum {
        BigNum(long n);
        BigNum(double n);
    }
    BigNum bn1(1.0);                    //这样也会引发对 BigNum 构造函数的重载解析
    ```
- std::auto_ptr 依赖了 C++中非常偏僻的重载解析规则。
- 最后记住一点，也是最重要的一点：大多数的时候，重载解析规则都会产生符合直观的结果。

= 完 结 =